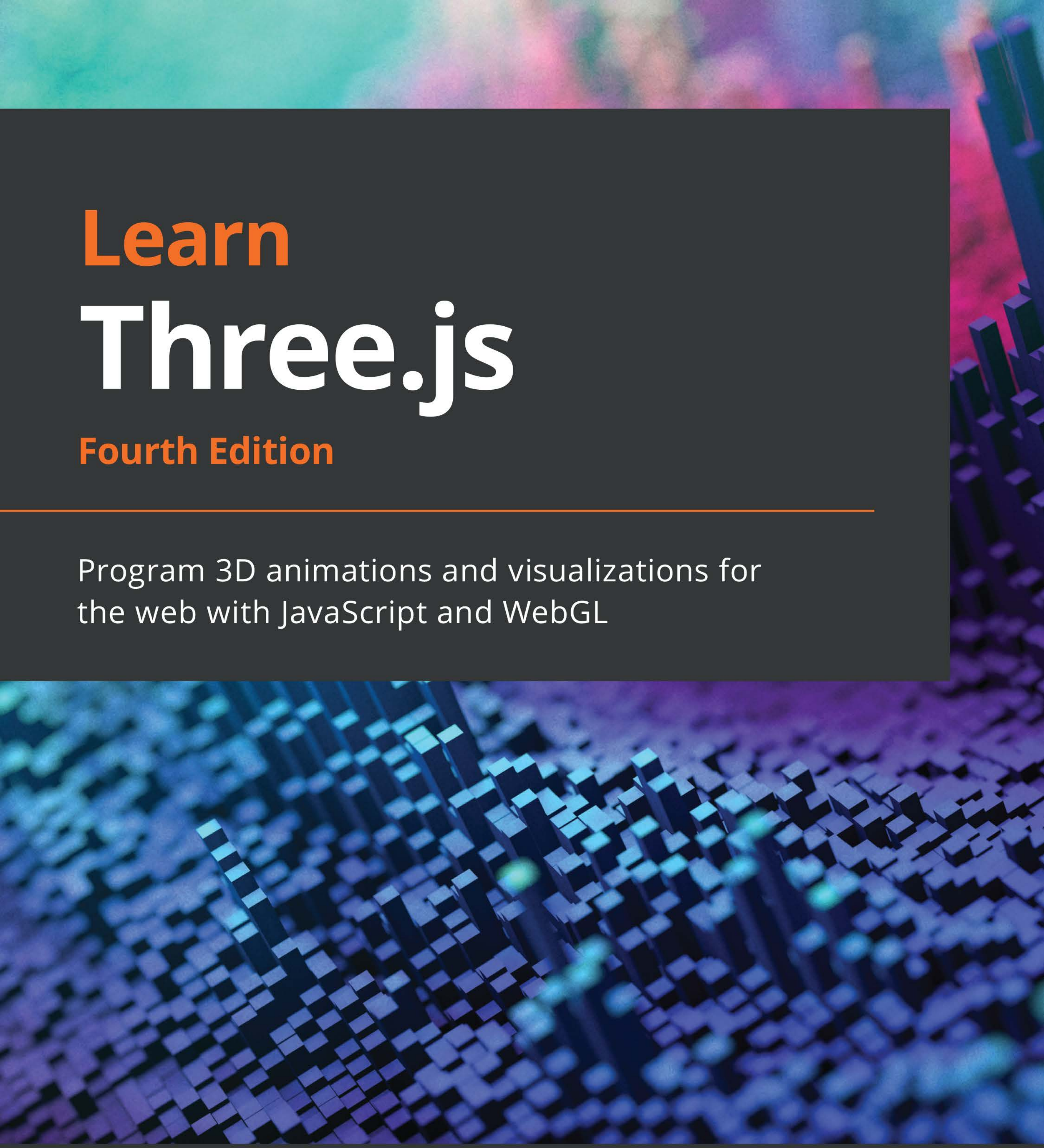


Learn Three.js

Fourth Edition

Program 3D animations and visualizations for the web with JavaScript and WebGL

Jos Dirksen

Learn Three.js

Fourth Edition

Program 3D animations and visualizations for the web with JavaScript and WebGL

Jos Dirksen

packt

BIRMINGHAM—MUMBAI

Learn Three.js

Fourth Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Pavan Ramchandani
Publishing Product Manager: Bhavya Rao
Senior Editor: Mark Dsouza
Senior Content Development Editor: Rashi Dubey
Technical Editor: Simran Udasi
Copy Editor: Safis Editing
Project Coordinator: Sonam Pandey
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Vijay Kamble
Marketing Coordinator: Anamika Singh

First published: October 2013
Second edition: March 2015
Third edition: August 2018
Fourth edition: February 2023
Production reference: 1170123

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80323-387-1
`www.packt.com`

To the memory of my father, who has been the best dad to me, an awesome grandfather to Sophie and Amber, and the kindest person I've ever known.

– Jos Dirksen

Contributors

About the author

Jos Dirksen has worked as a software developer and architect for almost two decades. He has a lot of experience with a large range of technologies, ranging from backend technologies, such as Java, Kotlin, and Scala, to frontend development using HTML5, CSS, JavaScript, and TypeScript. Besides working with these technologies, Jos also regularly speaks at conferences and likes to write about new and interesting technologies on his blog. He also likes to experiment with new technologies and see how they can best be used to create beautiful data visualizations.

He is currently working as a freelance full-stack engineer on various Scala and TypeScript projects.

Previously, Jos worked in many different roles in the private and public sectors, ranging from private companies such as ING, ASML, Malmberg, and Philips to organizations in the public sector, such as the Department of Defense and the Port of Rotterdam.

I want to thank the people who have supported me during writing this book. Especially my wife Brigitte, for always being there for me, and my mother Gerdie, who is always there to help us out!

About the reviewers

George Oscar Eugene Campbell has been in the industry since 2017 as a JavaScript developer. He's experienced in Three.js, Next.js, and TypeScript. He loves the natural creativity that programming offers and exploring the artistic side of things.

John Cotterell has been working as a frontend creative developer for 20 years now. He specializes in Three.js development, as well as AR, VR, and accompanying web technologies. More recently, he has spent time finding his way around the web audio API, GLSL, and the newer WebGPU standard, and is currently working on his first commercial game.

He lives in Bedford, UK, and when he's not playing or developing games, he likes to spend his free time visiting art galleries with his daughter, Shannon.

Mrunal Sawant is a highly proficient individual who has worked extensively with Three.js to develop different visualization projects.

He completed his master's in industrial mathematics with computer applications.

He is a skilled professional in the area of 3D visualization, CAD/BIM, and data interoperability (CAD data exchange).

Mrunal highly values his career and lives up to his motto "Work hard and play hard" by staying active doing sports and traveling around the country.

Table of Contents

Preface xv

Part 1: Getting Up and Running

1

Creating Your First 3D Scene with Three.js

Creating Your First 3D Scene with Three.js 3

Technical requirements 7

Getting the source code 9

Testing and experimenting with the examples 11

Exploring the HTML structure for Three.js applications 14

Rendering and viewing a 3D object 15

Setting up the scene 16

Adding lights 19

Adding the meshes 20

Adding an animation loop 22

Using lil-gui to control properties and make experimenting easier 28

Helper objects and util functions 30

Summary 31

2

The Basic Components that Make up a Three.js Application

The Basic Components that Make up a Three.js Application 33

Creating a scene 33

The basic functionality of a scene 34

How geometries and meshes are related 45

The properties and functions of a geometry 45

Functions and attributes for meshes 52

Using different cameras for different scenes 57

An orthographic camera versus a perspective camera 57

Looking at specific points 63

Debugging what a camera looks at 64

Summary 66

3

Working with Light Sources in Three.js 67

What lighting types are provided in Three.js? 68

Working with basic lights 68

THREE.AmbientLight 69

THREE.SpotLight 72

THREE.PointLight 80

THREE.DirectionalLight 84

Using the THREE.Color object 87

Working with special lights 89

THREE.HemisphereLight 90

THREE.RectAreaLight 91

THREE.LightProbe 93

THREE.LensFlare 96

Summary 100

Part 2: Working with the Three.js Core Components

4

Working with Three.js Materials 103

Understanding common material properties 104

Basic properties 105

Blending properties 106

Advanced properties 106

Starting with simple materials 107

THREE.MeshBasicMaterial 108

THREE.MeshDepthMaterial 112

Combining materials 114

THREE.MeshNormalMaterial 116

Multiple materials for a single mesh 119

Advanced materials 123

THREE.MeshLambertMaterial 124

THREE.MeshPhongMaterial 127

THREE.MeshToonMaterial 128

THREE.MeshStandardMaterial 129

THREE.MeshPhysicalMaterial 131

THREE.ShadowMaterial 132

Using your own shaders with THREE.ShaderMaterial 133

Customizing existing shaders with CustomShaderMaterial 139

Materials you can use for a line geometry 140

THREE.LineBasicMaterial 141

THREE.LineDashedMaterial 144

Summary 145

5

Learning to Work with Geometries 147

2D geometries 148
THREE.PlaneGeometry 149
THREE.CircleGeometry 151
THREE.RingGeometry 153
THREE.ShapeGeometry 155
3D geometries 159
THREE.BoxGeometry 160
THREE.SphereGeometry 161
THREE.CylinderGeometry 164
THREE.ConeGeometry 166
THREE.TorusGeometry 167
THREE.TorusKnotGeometry 169
THREE.PolyhedronGeometry 171
THREE.IcosahedronGeometry 173
THREE.TetrahedronGeometry 174
THREE.OctahedronGeometry 175
THREE.DodecahedronGeometry 176
Summary 178

6

Exploring Advanced Geometries 179

Learning advanced geometries 180
THREE.ConvexGeometry 180
THREE.LatheGeometry 182
BoxLineGeometry 184
THREE.RoundedBoxGeometry 185
TeapotGeometry 186
Creating a geometry by extruding a 2D shape 187
THREE.ExtrudeGeometry 187
THREE.TubeGeometry 190
Extruding 3D shapes from an SVG element 192
THREE.ParametricGeometry 195
Geometries you can use for debugging 199
THREE.EdgesGeometry 199
THREE.WireFrameGeometry 200
Creating a 3D text mesh 201
Rendering text 202
Adding custom fonts 204
Creating text using the Troika library 205
Summary 207

7

Points and Sprites 209

Understanding points and sprites 210
Styling particles using textures 215
Drawing an image on the canvas 215
Using textures to style particles 219
Working with sprite maps 226

Creating THREE.Points from existing geometry 228
Summary 230

Part 3: Particle Clouds, Loading and Animating Models

8

Creating and Loading Advanced Meshes and Geometries 233

Geometry grouping and merging 233
- Grouping objects together 233
- Merging geometries 238

Loading geometries from external resources 240
- Saving and loading in Three.js JSON format 243

Importing from 3D file formats 248
- The OBJ and MTL formats 248
- Loading a gLTF model 252
- Showing complete LEGO models 254
- Loading voxel-based models 256
- Showing proteins from PDB 258
- Loading a point cloud from a PLY model 262
- Other loaders 264

Summary 265

9

Animation and Moving the Camera 267

Basic animations 267
- Simple animations 268
- Selecting and moving objects 271
- Animating with Tween.js 277

Working with the camera 283
- ArcballControls 283
- TrackBallControls 286
- FlyControls 289
- FirstPersonControls 290
- OrbitControls 292

Morphing and skeleton animation 294
- Animation with morph targets 296
- Animation with a mixer and morph targets 297
- Animation using bones and skinning 304

Creating animations using external models 308
- Using gltfLoader 308
- Visualizing motions captured models using fbxLoader 310
- Loading an animation from a Quake model 313
- Loading an animation from a COLLADA model 314
- Visualizing a skeleton with BVHLoader 316

Summary 317

10

Loading and Working with Textures 319

Using textures in materials 319
Loading a texture and applying it to a mesh 320
Using a bump map to provide extra details to a mesh 324
Achieving more detailed bumps and wrinkles with a normal map 326
Using a displacement map to alter the position of vertices 329
Adding subtle shadows with an ambient occlusion map 331
Creating fake lighting using a lightmap 334
Metalness and roughness maps 336
Using an alpha map to create transparent models 339
Using an emissive map for models that glow 342
Using a specular map to determine shininess 344
Creating fake reflections using an environment map 347
Repeat wrapping 355

Rendering to a canvas and using it as a texture 357
Using the canvas as a color map 357
Using the canvas as a bump map 359
Using the output from a video as a texture 362

Summary 363

Part 4: Post-Processing, Physics, and Sounds

11

Render Postprocessing 367

Setting up Three.js for postprocessing 367
Creating THREE.EffectComposer 369
Configuring THREE.EffectComposer for postprocessing 370
Updating the render loop 370

Postprocessing passes 372
Simple postprocessing passes 373
Showing the output of multiple renderers on the same screen 380
Additional simple passes 381

Advanced EffectComposer flows using masks 382
Advanced pass – bokeh 387
Advance pass – ambient occlusion 390

Using THREE.ShaderPass for custom effects 392
Simple shaders 394
Blurring shaders 397

Creating custom postprocessing shaders 399
Custom grayscale shader 400
Creating a custom bit shader 404

Summary 406

12

Adding Physics and Sounds to Your Scene 409

Creating a basic Three.js scene with Rapier 410
- Setting up the world and creating the descriptions 412
- Rendering the scene and simulating the world 414

Simulating dominos in Rapier 415

Working with restitution and friction 421

Rapier-supported shapes 425

Using joints to limit the movement of objects 428
- Connecting two objects with a fixed joint 428
- Connecting objects with a spherical joint 431
- Limiting rotation with a revolute joint 433
- Limiting movement to a single axis with a prismatic joint 437

Adding sound sources to your scene 438

Summary 441

13

Working with Blender and Three.js 443

Exporting from Three.js and importing into Blender 444

Exporting a static scene from Blender and importing it into Three.js 450

Exporting an animation from Blender and importing it into Three.js 454

Baking lightmaps and ambient occlusion maps in Blender 464
- Setting up a scene in Blender 464
- Adding lighting to the scene 466
- Baking the light textures 469
- Exporting the scene and importing it into Blender 475
- Baking an ambient occlusion map in Blender 478

Custom UV modeling in Blender 480

Summary 486

14

Three.js Together with React, TypeScript, and Web-XR 487

Using Three.js with TypeScript 488
Using Three.js and React with TypeScript 493
Using Three.js and React with React Three Fiber 499
Three.js and VR 508
Three.js and AR 514
Summary 516

Index 517

Other Books You May Enjoy 530

Preface

Three.js has become the standard way of creating stunning 3D WebGL content over the last couple of years. In this edition, we'll look at all the features of Three.js and provide additional content on how to integrate Three.js with Blender, React, TypeScript, and the newest physics engine.

In this book, you'll learn how to create and animate immersive 3D scenes directly in your browser using the full potential of WebGL and modern browsers.

The book starts with the basic concepts and building blocks used in Three.js and helps you explore these essential topics in detail through extensive examples and code samples. You'll also learn how to create realistic-looking 3D objects using textures and materials. Besides creating these objects manually, we'll also explain how to load existing models from an external source. Next, you'll understand how to easily control the camera using the Three.js built-in camera controls, which will enable you to fly or walk around the 3D scene you've created. Later chapters will then show you how to use the HTML5 video and canvas elements as materials for your 3D objects and animate your models. You will learn how to use morph and skeleton-based animation, before understanding how to add physics, such as gravity and collision detection, to your scene. Finally, we'll explain how to combine Blender with Three.js, how to integrate Three.js with React and TypeScript, and how you can use Three.js to create VR and AR scenes.

By the end of this book, you'll have gained the skills you need to create 3D-animated graphics using Three.js.

Who this book is for

This book is for JavaScript developers who are looking to learn how to use the Three.js library confidently.

What this book covers

Chapter 1, *Creating Your First 3D Scene with Three.js*, introduces the Three.js library, explains how to set up your development environment, and shows how to create your first application.

Chapter 2, *The Basic Components that Make up a Three.js Application*, explains the core concepts behind Three.js and introduces the basic components you'll need for each Three.js application.

Chapter 3, *Working with Light Sources in Three.js*, goes through all the available light sources you can use in Three.js to add lights to your Three.js scenes.

Chapter 4, *Working with Three.js Materials*, shows an overview of the different materials you can use when creating the objects you're visualizing with Three.js.

Chapter 5, *Learning to Work with Geometries*, explains which basic geometries are provided with Three.js and how you can use and customize them.

Chapter 6, *Exploring Advanced Geometries*, goes into detail explaining the advanced geometries provided by Three.js.

Chapter 7, *Points and Sprites*, describes how you can create and manipulate a large number of points and sprites on the screen, and how to change what these points look like.

Chapter 8, *Creating and Loading Advanced Meshes and Geometries*, gives examples and instructions on how to group and merge objects together, and how to load existing models from external sources.

Chapter 9, *Animations and Moving the Camera*, talks about the different camera controls provided by Three.js, and explains how animations work in Three.js and how to load animations from external sources.

Chapter 10, *Loading and Working with Textures*, gives an overview of the different types of textures that are available in Three.js, and how you can use them to configure the Three.js materials.

Chapter 11, *Render Postprocessing*, describes how you can add post-rendering effects such as blur, bloom, and colorify to your scene.

Chapter 12, *Adding Physics and Sounds to Your Scene*, introduces the Rapier physics library, which allows you to have objects interact with each other simulating the real world. This chapter also shows how you can add positional sound to your scenes.

Chapter 13, *Working with Blender and Three.js*, provides information on how you can integrate Blender with Three.js, exchange models, and use Blender to bake specific textures.

Chapter 14, *Three.js Together with React, TypeScript, and Web-XR*, shows how you can combine Three.js with React and TypeScript, and gives an overview of the support Three.js has for the Web-XR standard to create VR and AR scenes.

To get the most out of this book

You don't need to know much before starting with this book. The only requirement is that you have some basic knowledge of JavaScript. This book provides instructions on how to set up your development environment, install any additional tools and libraries, and run the examples.

Software covered in the book	**Operating system requirements**
Three.js r147	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Learn-Three.js-Fourth-edition`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "You can do these steps yourself, or look in the `three-ts` folder and just run `yarn install` there to skip this setup."

A block of code is set as follows:

```
import './style.css'
import { initThreeJsScene } from './threeCanvas'
const mainElement = document.
querySelector<HTMLDivElement>('#app')
if (mainElement) {
  initThreeJsScene(mainElement)
}
```

Any command-line input or output is written as follows:

```
$ git --version
git version 2.30.1 (Apple Git-130)
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "For this, we need to create a new debug configuration from the **Run | Edit Configurations** menu."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803233871`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1:
Getting Up and Running

In the first part, we'll introduce the basic concepts of Three.js and make sure you've got all the information to get up and running. We'll start by setting up your development environment, and in the following chapters, explore some of the core concepts that make up Three.js.

In this part, there are the following chapters:

- *Chapter 1, Creating Your First 3D Scene with Three.js*
- *Chapter 2, The Basic Components that Make up a Three.js Application*
- *Chapter 3, Working with Light Sources in Three.js*

1

Creating Your First 3D Scene with Three.js

In recent years, modern browsers have acquired powerful features that can be accessed directly from **JavaScript**. You can easily add video and audio with **HTML5 tags** and create interactive components through the use of **HTML5 Canvas**. Together with HTML5, modern browsers also support **WebGL**. With WebGL, you can directly make use of the processing resources of your graphics card and create high-performance 2D and 3D computer graphics. Using WebGL directly from JavaScript to create and animate 3D scenes is a very complex, verbose, and error-prone process. **Three.js** is a library that makes this a lot easier. The following list shows some of the things that are very easy to do with Three.js:

- Create simple and complex 3D geometries and render them in any browser
- Animate and move objects through a 3D scene
- Apply textures and materials to your objects
- Use different light sources to illuminate the scene
- Use models from 3D modeling software and export generated models into these programs
- Add advanced post-processing effects to your 3D scene
- Create and work with custom shaders
- Create, visualize, and animate point clouds
- Create **virtual reality** (**VR**) and **augmented reality** (**AR**) scenes

With a couple of lines of JavaScript (or **TypeScript**, as we'll see later in this book), you can create anything, from simple 3D models to photorealistic scenes, all of which are rendered in real time in the browser. For instance, *Figure 1.1* shows an example of what can be done with Three.js (you can see the animation for yourself by opening `https://threejs.org/examples/webgl_animation_keyframes.html` in your browser):

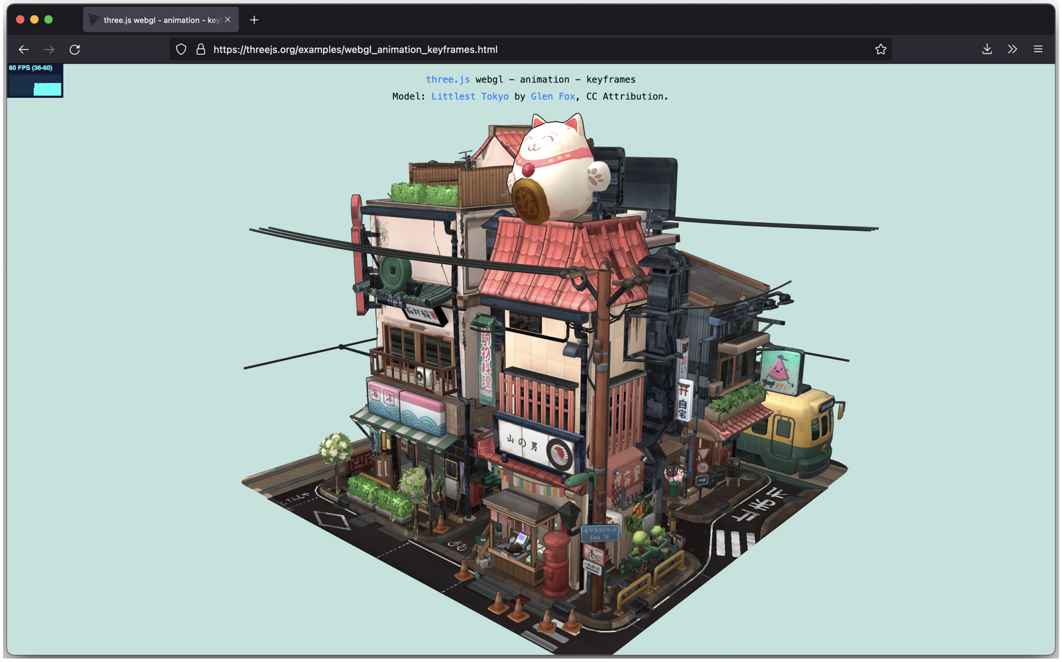

three.js webgl - animation - key
https://threejs.org/examples/webgl_animation_keyframes.html
60 FPS (36-60)
three.js webgl - animation - keyframes
Model: Littlest Tokyo by Glen Fox, CC Attribution.

Figure 1.1 – Three.js rendered and animated scene

In this chapter, we’ll directly dive into Three.js and create a couple of examples that will show you how Three.js works, and that you can use to play around with and get to know Three.js a little bit. We won’t dive into all the technical details yet; you'll learn about those in the following chapters. By the end of this chapter, you’ll be able to create a scene, and run and explore all the examples in this book.

We’ll start this book with a short introduction to Three.js and then quickly move on to the first few examples and code samples. Before we get started, let’s quickly look at the most important browsers out there and their support for WebGL (and WebGPU).

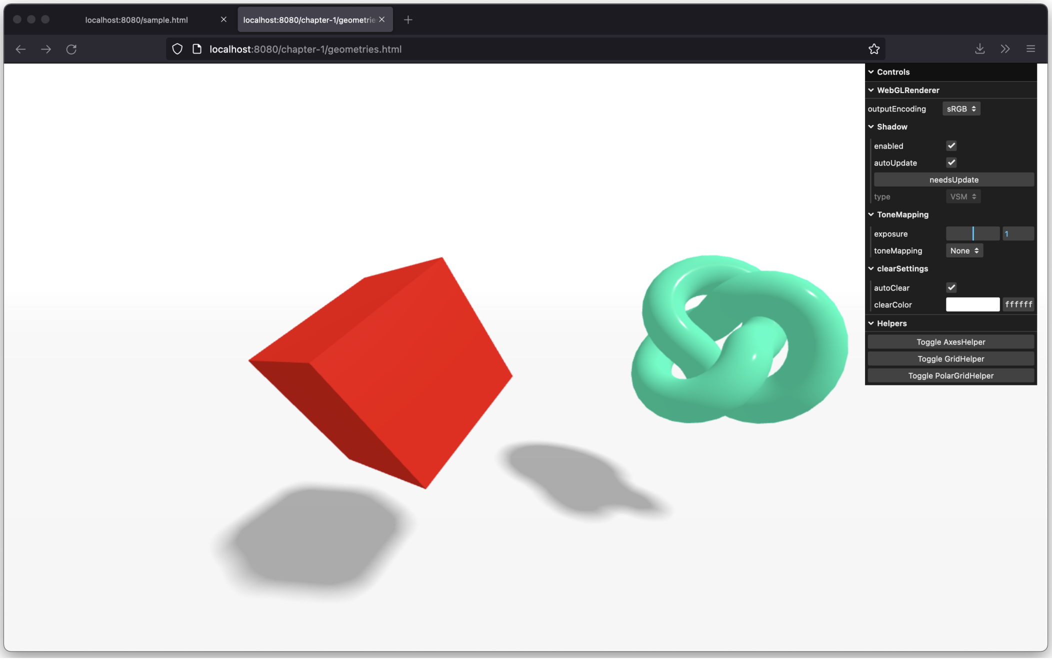
Note

All modern browsers on desktop, as well as on mobile, currently support WebGL. Older versions of IE (those before version 11) won't be able to run WebGL-based applications. On mobile, most browsers support WebGL. So, with WebGL, you can create interactive 3D visualizations that run very well on desktops, as well as on mobile devices.

In this book, we'll focus on the WebGL-based renderer provided by Three.js. There is, however, also a CSS 3D-based renderer, which provides an easy API to create CSS 3D-based 3D scenes. A big advantage of using a CSS 3D-based approach is that this standard is supported on all mobile and desktop browsers and allows you to render HTML elements in a 3D space. We won't go into the details of this browser but will show an example in *Chapter 7*, *Points and Sprites*.

Besides WebGL, a new standard for using the GPU to render in your browser called WebGPU is being developed, which will provide even better performance than WebGL and, in the future, become the new standard. When you use Three.js, you don't have to worry about this change. Three.js already partly supports WebGPU and as that standard matures, so will the support of this standard in Three.js. So, everything you create with Three.js will also work out of the box with WebGPU.

In this first chapter, you'll directly create a 3D scene and be able to run it on a desktop or mobile device. We'll explain the core concepts of Three.js, and if there are more advanced topics, we'll mention in what chapter we'll explain these in more detail. In this chapter, we'll create two different scenes. The first one will show a basic geometry rendered in Three.js, as shown in the following figure:


localhost:8080/sample.html
localhost:8080/chapter-1/geometrie
localhost:8080/chapter-1/geometries.html
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

outputEncoding sRGB
enabled ☑
autoUpdate ☑
type VSM
exposure 1
toneMapping None
autoClear ☑
clearColor ffffff

Figure 1.2 – Default geometries rendered

After that, we'll also quickly show you how you can load external models, and how easy it is to create realistic-looking scenes. The result of the second example will look like this:

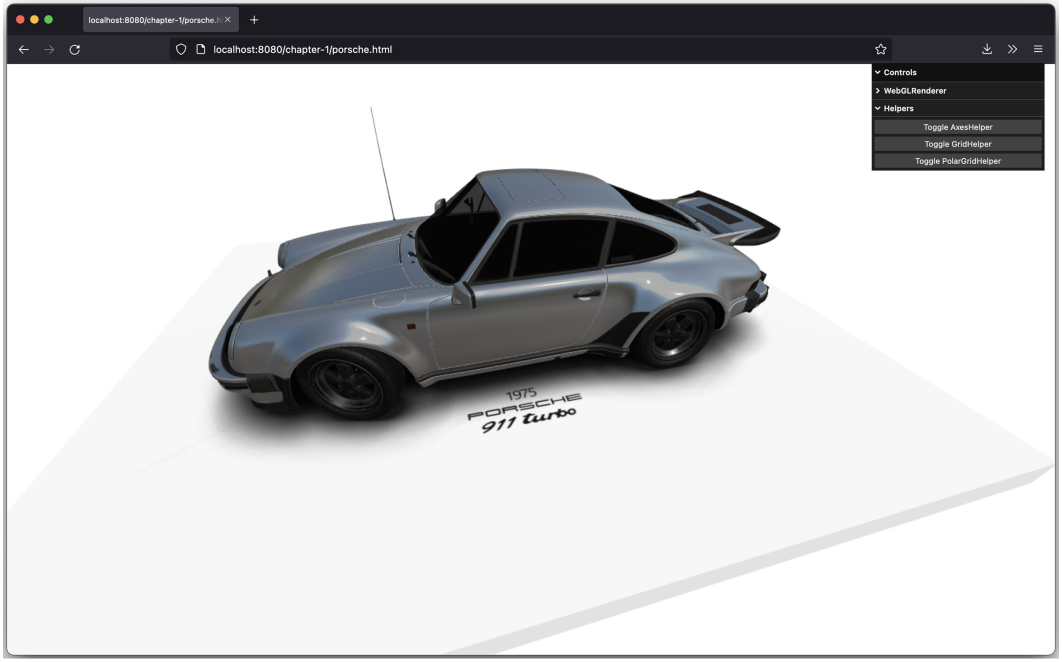

localhost:8080/chapter-1/porsche.h
localhost:8080/chapter-1/porsche.html
Controls
WebGLRenderer
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper
1975
PORSCHE
911 turbo

Figure 1.3 – Rendering an externally loaded model

Before you start working on these examples, in the next couple of sections, we'll look at the tools you need to easily work with Three.js and how you can download the examples shown in this book.

In this chapter, we'll cover the following topics:

- Requirements for using Three.js
- Downloading the source code and examples used in this book
- Testing and experimenting with the examples
- Rendering and viewing a 3D object
- Introducing a couple of helper libraries for statistics and controlling the scene

Technical requirements

Three.js is a JavaScript **library**, so all you need to create Three.js WebGL applications is a text editor and one of the supported browsers to render the results. I would like to recommend the following text editors, which I've used extensively over the last couple of years for various projects:

- **Visual Studio Code**: This free editor from Microsoft runs on all major platforms and provides great syntax highlighting and smart completion based on types, function definitions, and imported libraries. It provides a very clean interface and is great for working on JavaScript projects. It can be downloaded from here: `https://code.visualstudio.com/`. If you don't want to download this editor, you can also just navigate to `https://vscode.dev/`, which will launch an editor directly in your browser, from which you can connect to a GitHub repository or access directories on your local filesystem.
- **WebStorm**: This editor from *JetBrains* offers great support for editing JavaScript. It supports code completion, automatic deployment, and JavaScript debugging, directly from the editor. Besides this, WebStorm has excellent GitHub (and other version control system) support. You can download a trial edition from `http://www.jetbrains.com/webstorm/`.
- **Notepad++**: Notepad++ is a general-purpose editor that supports code highlighting for a wide range of programming languages. It can easily lay out and format JavaScript. Note that Notepad++ is only for Windows. You can download Notepad++ from `http://notepad-plus-plus.org/`.
- **Sublime Text Editor**: Sublime is a great editor that offers very good support for editing JavaScript. Besides this, it provides many very helpful selections (such as multi-line select) and edit options, which, once you get used to them, provide a really good JavaScript-editing environment. Sublime can also be tested for free and can be downloaded from `http://www.sublimetext.com/`.

Even if you don't use any of these editors, there are a lot of editors available, both open source and commercial, that you can use to edit JavaScript and create your Three.js projects, since all you need is the ability to edit text. An interesting project you might want to look at is AWS Cloud9 (`http://c9.io`). This is a cloud-based JavaScript editor that can be connected to a GitHub account. This way, you can directly access all the source code and examples from this book and experiment with them.

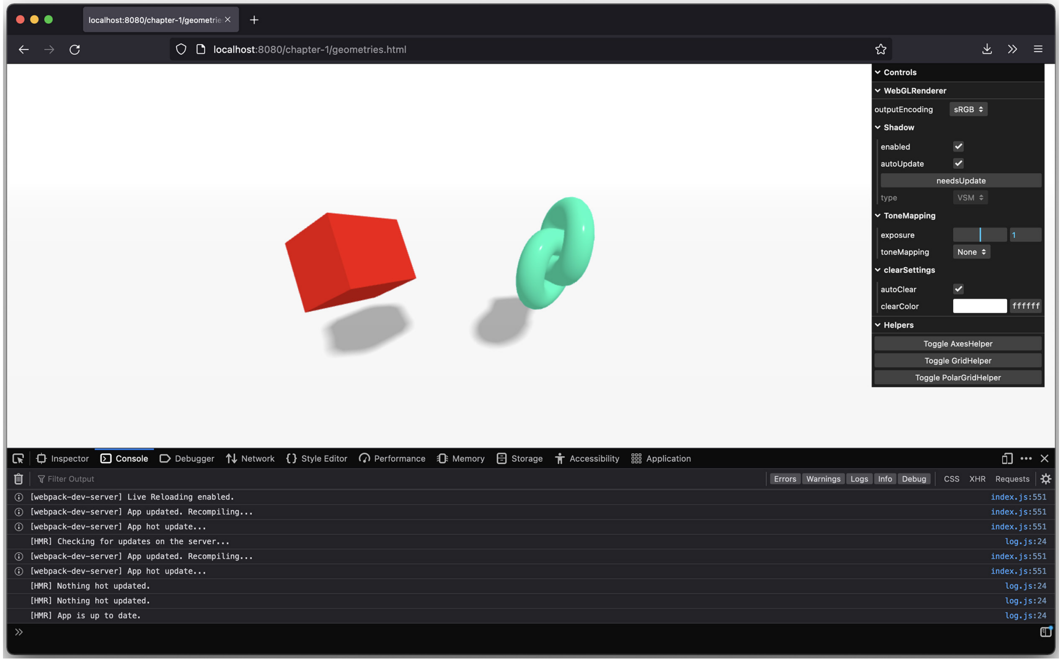
Note

Besides these text-based editors, which you can use to edit and experiment with the sources from this book, Three.js currently also provides an online editor.

With this editor, which you can find at `http://threejs.org/editor/`, you can create Three.js scenes using a graphical approach.

I suggest picking up Visual Studio Code. It is a very lightweight editor with great support for JavaScript and has several other extensions that make writing JavaScript applications easier.

Earlier, I mentioned that most modern web browsers support WebGL and can be used to run Three.js examples. I usually run my code in Firefox. The reason is that, often, Firefox has the best support and performance for WebGL and it has a great JavaScript debugger. With this debugger, as shown in the following screenshot, you can quickly pinpoint problems using, for instance, breakpoints and console output:


localhost:8080/chapter-1/geometrie
localhost:8080/chapter-1/geometries.html
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Filter Output
Errors
Warnings
Logs
Info
Debug
CSS
XHR
Requests

outputEncoding: sRGB
enabled: ☑
autoUpdate: ☑
type: VSM
exposure: 1
toneMapping: None
autoClear: ☑
clearColor: ffffff

```
[webpack-dev-server] Live Reloading enabled.          index.js:551
[webpack-dev-server] App updated. Recompiling...      index.js:551
[webpack-dev-server] App hot update...                index.js:551
[HMR] Checking for updates on the server...           log.js:24
[webpack-dev-server] App updated. Recompiling...      index.js:551
[webpack-dev-server] App hot update...                index.js:551
[HMR] Nothing hot updated.                            log.js:24
[HMR] Nothing hot updated.                            log.js:24
[HMR] App is up to date.                              log.js:24
```

Figure 1.4 – Firefox debugger

Note

All the examples in this book will work just as well with Chrome as they do with Firefox. So, if that's your browser of choice, you can, of course, use that one instead.

Throughout this book, I'll give you pointers on debugger usage and other debugging tips and tricks. That's enough by way of an introduction for now; let's get the source code and start with the first scene.

Getting the source code

All the code for this book is available on GitHub (`https://github.com/PacktPublishing/Learn-Three.js-Fourth-edition`). GitHub is a site that hosts Git repositories. You can use these to store, access, and version source code. There are a couple of ways that you can get the sources for yourself. You can do either of the following:

- Clone the Git repository. This means you use the `git` command-line tool to get the latest version of the sources for this book.
- Download and extract an archive from GitHub, which contains everything.

In the following two subsections, we'll explore these options in a bit more detail.

Using git to clone the repository

One way to get all the examples is to *clone* this repository using the `git` command-line tool. To do this, you need to download a Git client for your operating system. If you've got an up-to-date operating system, you probably already have Git installed. You can quickly check this by running the following in a terminal:

```
$ git --version
git version 2.30.1 (Apple Git-130)
```

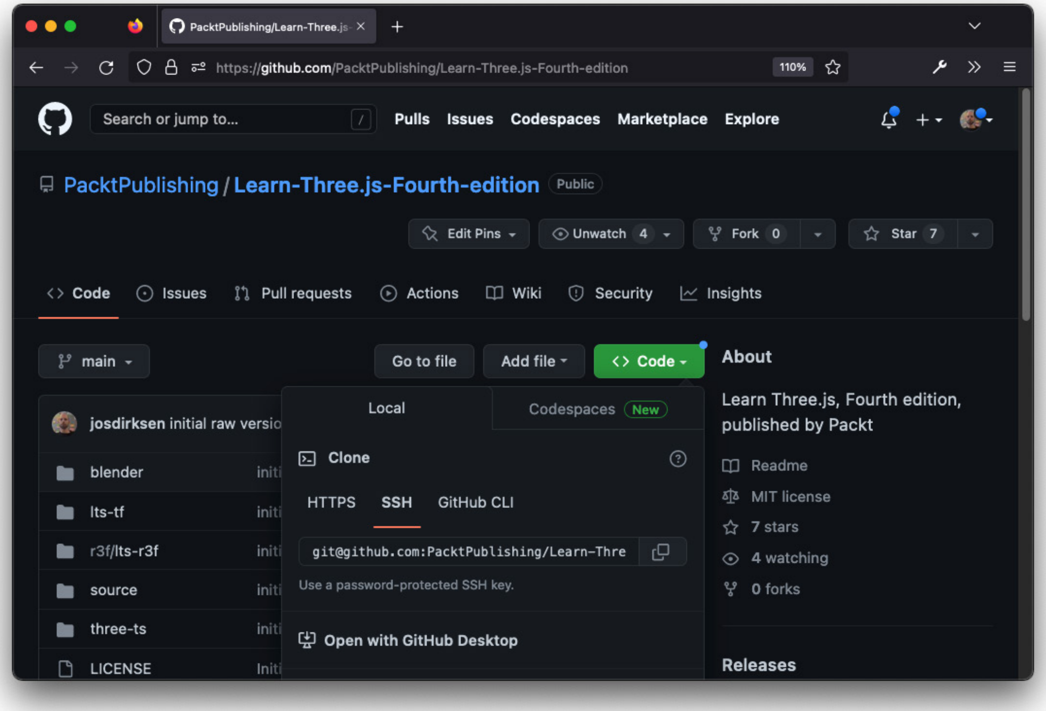
If the command isn't installed yet, you can get a client and instructions on how to install it from here: `http://git-scm.com`. After installing Git, you can use the `git` command-line tool to clone this book's repository. Open a command prompt and go to the directory where you want to download the sources. In that directory, run the following:

```
$ git clone https://github.com/PacktPublishing/Learn-Three.
js-Fourth-edition.
git clone git@github.com:PacktPublishing/Learn-Three.js-Fourth-
edition.git
Cloning into 'learning-threejs-fourth'...
remote: Enumerating objects: 96, done.
remote: Counting objects: 100% (96/96), done.
remote: Compressing objects: 100% (85/85), done.
fetch-pack: unexpected disconnect while reading sideband packet
...
```

After doing this, all the source code will be downloaded into the `learning-threejs-fourth` directory. From that directory, you can run all the examples explained throughout this book.

Downloading and extracting the archive

If you don't want to use `git` to download the sources directly from GitHub, you can also download an archive. Open `https://github.com/PacktPublishing/Learn-Three.js-Fourth-edition` in a browser and click on the **Code** button on the right-hand side. This will give you the option to download all the sources in a single ZIP file by clicking on the **Download ZIP** option:


PacktPublishing/Learn-Three.js
https://github.com/PacktPublishing/Learn-Three.js-Fourth-edition
110%
Search or jump to...
Pulls
Issues
Codespaces
Marketplace
Explore
PacktPublishing / Learn-Three.js-Fourth-edition
Public
Edit Pins
Unwatch 4
Fork 0
Star 7
Code
Issues
Pull requests
Actions
Wiki
Security
Insights
main
Go to file
Add file
Code
josdirksen initial raw versio
blender
lts-tf
r3f/lts-r3f
source
three-ts
LICENSE
Local
Codespaces New
Clone
HTTPS
SSH
GitHub CLI
Open with GitHub Desktop
About
Readme
MIT license
7 stars
4 watching
0 forks
Releases

git@github.com:PacktPublishing/Learn-Thre

Use a password-protected SSH key.

Learn Three.js, Fourth edition, published by Packt

Figure 1.5 – Downloading the archive from GitHub

After extracting this to a directory of your choice, all the examples will become available.

Note

You can also download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

Now that you've downloaded or cloned the source code, let's quickly check whether everything is working and familiarize you with the directory structure.

Testing and experimenting with the examples

The code and examples are organized per chapter and, with the examples, we will provide a simple integrated server that you can use to access all the examples. To get this server up and running, we need to install *Node.js* and *npm*. These tools are used to manage JavaScript packages and build JavaScript applications and make it easier to modularize our Three.js code and integrate existing JavaScript libraries.

To install these two tools, go to `https://nodejs.org/en/download/` and select the appropriate installer for your operating system. Once installed, open a terminal and check whether everything is working. On my machine, the following versions are being used:

```
$ npm --version
8.3.1
$ node --version
v16.14.0
```

Once these tools have been installed, we need to perform a few steps to get all the externally needed dependencies before we can build and access the examples:

1. First, we need to download the external libraries used in the examples. For instance, Three.js is one of the dependencies we need to download.

   To download all the dependencies, run the following command in the directory where you downloaded or extracted all the examples:

   ```
   $ npm install
   added 570 packages, and audited 571 packages in 21s
   ```

   The preceding command will start downloading all the required JavaScript libraries and store these in the `node_modules` folder.

2. Next, we need to build the examples. Doing so will combine our source code and the external libraries into a single file, which we can show in the browser.

   To build the examples using `npm`, use the following command:

   ```
   $ npm run build
   > ltjs-fourth@1.0.0 build
   > webpack build
   ...
   ```

   Note that you only have to run the two preceding commands once.

3. With that, all the examples will have been built and are ready for you to explore. To open these examples, you need a web server. To start a server, simply run the following command:

```
$ npm run serve
> ltjs-fourth@1.0.0 serve
> webpack serve -open
<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:8080/
<i> [webpack-dev-server] On Your Network (Ipv4):
http://192.168.68.144:8080/
<i> [webpack-dev-server] On Your Network (Ipv6): http://
[fe80::1]:8080/
…
```

At this point, you'll probably notice that `npm` has already opened your default browser and shows the content of `http://localhost:8080` (if this isn't the case, just open your browser of choice and navigate to `http://localhost:8080`). You'll be presented with an overview of all the chapters. In each of these subfolders, you'll find the examples that are explained in that chapter:

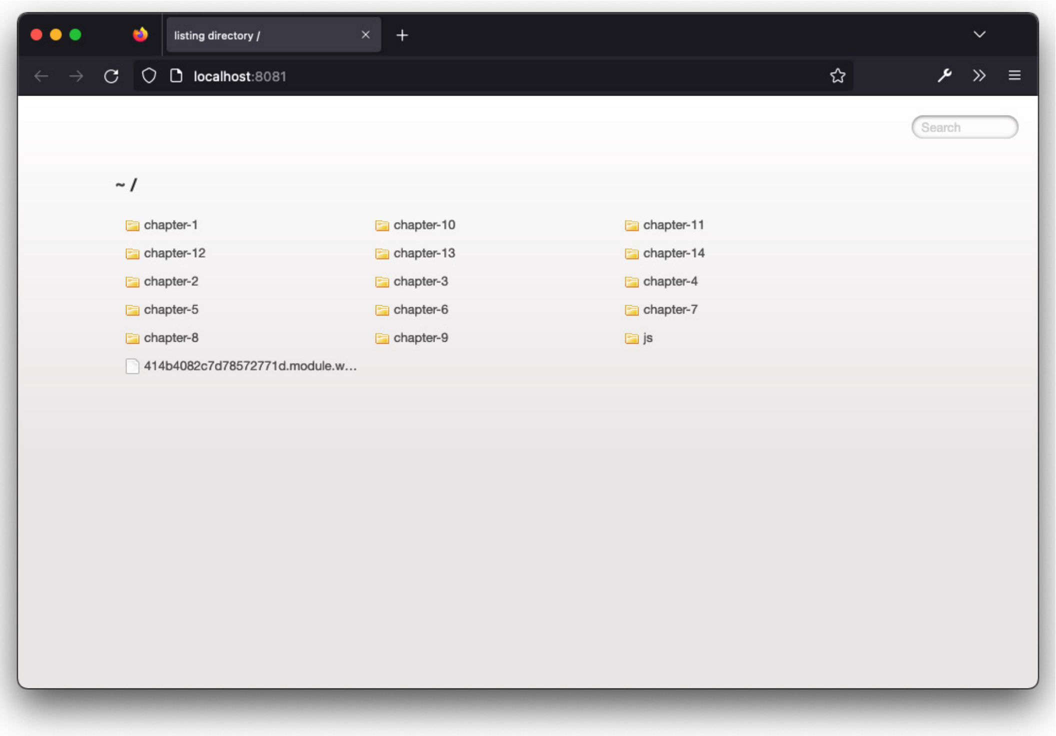

listing directory /
localhost:8081
Search
~ /
chapter-1
chapter-10
chapter-11
chapter-12
chapter-13
chapter-14
chapter-2
chapter-3
chapter-4
chapter-5
chapter-6
chapter-7
chapter-8
chapter-9
js
414b4082c7d78572771d.module.w...

Figure 1.6 – Overview of all the chapters and examples

One very interesting feature of this server is that we can now see the changes we make to the source code immediately reflected in the browser. If you have started the server by running `npm run serve`, open up the `chapter-01/geometries.js` example from the sources you've downloaded in your editor and change something; you'll see that this is also changed at the same time in your browser after you have saved the change. This makes testing changes and fine-tuning colors and lights much easier. If you open the `chapter-01/geometries.js` file in your code editor, and you open the `http://localhost:8080/chapter-01/geometries.html` example in your browser, you can see this in action. In your editor, change the color of the cube. To do so, find the following code:

```
initScene(props)(({ scene, camera, renderer, orbitControls })
=> {
  const geometry = new THREE.BoxGeometry();
  const cubeMaterial = new THREE.MeshPhongMaterial({
    color: 0xFF0000,
  });
```

Change it to the following:

```
initScene(props)(({ scene, camera, renderer, orbitControls })
=> {
  const geometry = new THREE.BoxGeometry();
  const cubeMaterial = new THREE.MeshPhongMaterial({
    color: 0x0000FF,
  });
```

Now, when you save the file, you'll immediately see that the color of the cube in the browser changes, without you having to refresh the browser or do anything else.

Note

The setup we're working with in this book is one of many different approaches you can use to develop web applications. Alternatively, you can include Three.js (and other libraries) directly in your HTML file or use an approach with `import-maps`, as is done with the example on the Three.js website. All of these have advantages and disadvantages. For this book, we've chosen an approach that makes it easy to experiment with the sources and get direct feedback in the browser, and closely resembles how these kinds of applications are built normally.

A good starting point to understand how everything works together is by looking at the HTML file that we opened in the browser.

Exploring the HTML structure for Three.js applications

In this section, we'll look at the source of the `geometries.html` file. You can do this by looking at the source in the browser or opening the file from the `dist/chapter-1` folder in the same location where you downloaded the source for this book:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <style>
    body {
      margin: 0;
    }
  </style>
  <script defer src="../js/vendors-node_modules_three_
    build_three_module_js.js"></script>
  <script defer src="../js/vendors-node_modules_lil-gui_
    dist_lil-gui_esm_js.js"></script>
  <script defer src="../js/vendors-node_modules_three_
    examples_jsm_controls_OrbitControls_js.js"></script>
  <script defer src="../js/geometries.js"></script>
</head>
<body>
</body>
</html>
```

This code is generated when you run the `npm run build` step. This will combine all the sources and external libraries you've used into separate source files (called bundles) and add them to this page. So, you don't need to do this yourself. The first three `<script>` tags refer to any of the external libraries we use. Later in the book, we'll introduce other libraries such as **React.js** and **Tween.js**. Those will be included in the same manner automatically. The only other elements here are `<style>` and `<body>`. `<style>` is used to disable any margins in the page, so we can use the complete browser viewport to show our 3D scenes. Furthermore, we'll add the 3D scene programmatically into an empty `<body>` element, which we'll explain in the next section.

If you do want to add custom HTML elements here, you can, of course, do that. In the root of the downloaded code, you'll find a `template.html` file, which is used by the build process to create the individual HTML files for the examples. Anything you add there will be added to all the examples.

We won't dive too deep into how this works since that's outside the scope of this book. However, if you want to learn more about how this works, a couple of good resources on *webpack* (which we use for this) are as follows:

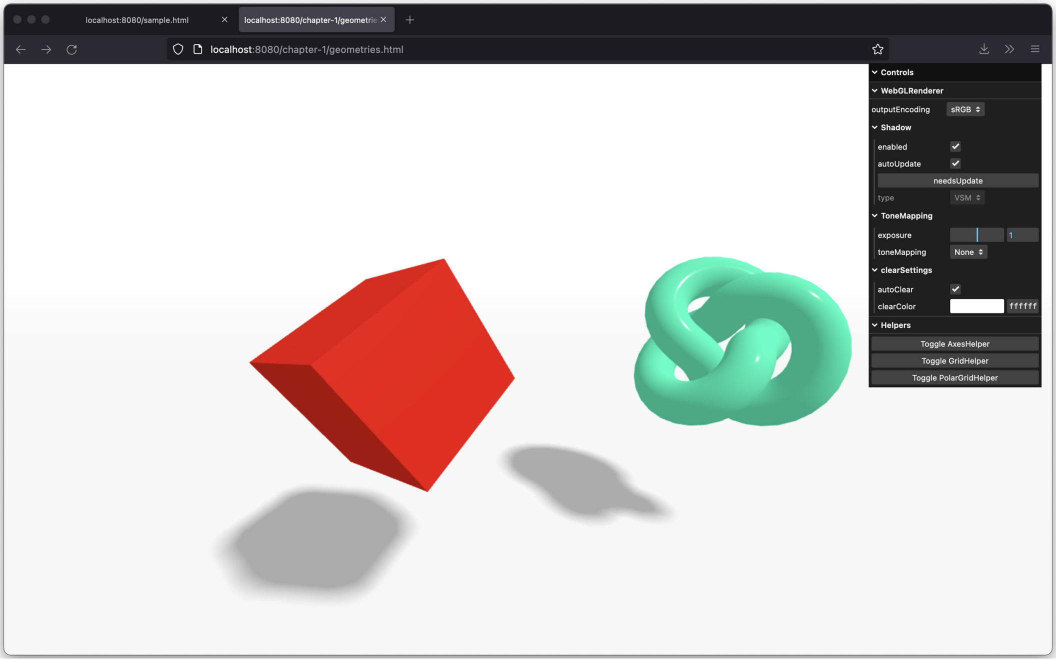
- The getting started with webpack guide: `https://webpack.js.org/guides/getting-started/`. This site contains a tutorial that explains the reason why we need webpack for JavaScript development, and how the basic concepts work.
- Information on the *HTML webpack plugin*: `https://github.com/jantimon/html-webpack-plugin`. Here, you can find information on the webpack plugin we use to combine the sources into the separate HTML pages you see when you open the browser after running `npm run build` and then running `npm run serve`.

Note that we don't have to explicitly initialize our scene or call JavaScript. Whenever we open this page and the `geometries.js` file is loaded, the JavaScript from that file will run and create our 3D scene.

Now that we've set up the basic structure, we can create and render our first scene.

Rendering and viewing a 3D object

In this section, you'll create your first scene, which is a simple 3D scene that looks like this:


localhost:8080/sample.html
localhost:8080/chapter-1/geometrie
localhost:8080/chapter-1/geometries.html
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

outputEncoding sRGB
enabled ☑
autoUpdate ☑
type VSM
exposure 1
toneMapping None
autoClear ☑
clearColor ffffff

Figure 1.7 – First scene with two standard geometries

In the preceding screenshot, you can see two objects that rotate. These objects are called *meshes*. A mesh describes the geometry of an object – that is, its shape – and contains information about the material of the object. A mesh determines how the shape gets shown on screen through traits such as color, or whether the object is shiny or transparent.

In the previous screenshot, we can identify three of these meshes:

Object	**Description**
Plane	This is a two-dimensional rectangle that serves as the ground area. In *Figure 1.7*, you can see this since it shows the shadows cast by the two meshes. We will create this as a very large rectangle so that you don't see any edges.
Cube	This is a three-dimensional cube and is shown on the left of *Figure 1.7*. It is rendered in *red*.
Torus knot	This is the *TorusKnot* you can see to the right of *Figure 1.7*. This one is rendered in *green*.

Figure 1.8 – Overview of the objects in the scene

To get all this on screen, we need to perform a couple of steps, which we'll explain in the upcoming sections.

Setting up the scene

Each Three.js application at least needs a camera, a scene, and a renderer. The scene is the container that holds all the objects (meshes, cameras, and lights), the camera determines what part of the scene is shown when it is rendered, and the renderer takes care of creating the output on the screen, taking into account all the information from the meshes, cameras, and lights in the scene.

All the code we'll be discussing can be found in the `chapter-1/getting-started.js` file. The basic structure of this file is as follows:

```
import * as THREE from "three";
import Stats from 'three/examples/jsm/libs/stats.module'
import { OrbitControls } from 'three/examples/jsm/controls/
OrbitControls'
// create a scene
...
// setup camera
...
// setup the renderer and attach to canvas
...
```

```
// add lights
...
// create a cube and torus knot and add them to the scene
...
// create a very large ground plane
...
// add orbitcontrols to pan around the scene using the
   mouse
...
// add statistics to monitor the framerate
...
// render the scene
```

If you look through the preceding steps, you might have noticed that a lot of these steps are the same for each scene you create. Since we've got a lot of examples in this book that show different features of Three.js, we'll extract this code into a couple of helper files. We'll show how we did this at the end of this chapter. For now, we will walk through the different steps and introduce you to the basic components of a Three.js scene.

First, we must create a `THREE.Scene`. This is a basic container that will hold all of the meshes, lights, and cameras and has a couple of simple properties, which we'll explore in more depth in the next chapter:

```
// basic scene setup
const scene = new THREE.Scene();
scene.backgroundColor = 0xffffff;
scene.fog = new THREE.Fog(0xffffff, 0.0025, 50);
```

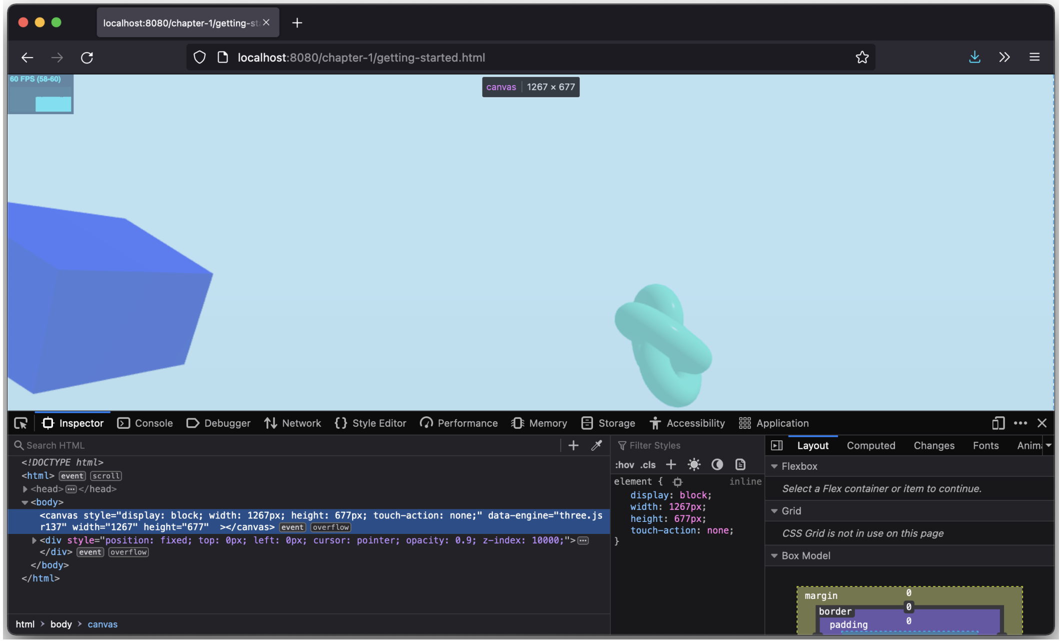
Here, we will create the container object that will hold all our objects, set the background color of this scene to white (`0xffffff`), and enable the fog effect in this scene. With fog enabled, objects further away from the camera will slowly get hidden by fog.

The next step is creating the camera and the renderer:

```
// setup camera and basic renderer
const camera = new THREE.PerspectiveCamera(
  75,
  window.innerWidth / window.innerHeight,
  0.1,
  1000
);
camera.position.x = -3;
camera.position.z = 8;
camera.position.y = 2;
// setup the renderer and attach to canvas
const renderer = new THREE.WebGLRenderer({ antialias: true
  });
renderer.outputEncoding = THREE.sRGBEncoding;
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.VSMShadowMap;
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.setClearColor(0xffffff);
document.body.appendChild(renderer.domElement);
```

In the preceding code, we created a `PerspectiveCamera`, which determines what part of the scene is rendered. Don't worry too much about the parameters at this point, since we'll discuss those in detail in *Chapter 3*, *Working with Light Sources in Three.js*. We also positioned the camera at the specified *x*-, *y*-, and *z*-oordinates. The camera will, by default, look at the center of the scene (which is `0`, `0`, `0`), so we don't need to change anything for that.

In this code fragment, we also created a `WebGLRenderer`, which we will use to render the view from the camera on the scene. Ignore the other properties for now; we'll explain these in the next few chapters when we dive into the details of `WebGLRenderer` and how you can fine-tune the colors and work with shadows. One interesting part to notice is `document.body.appendChild(renderer.domElement)`. This step adds an HTML `canvas` element to the page, which shows the output of the renderer. You can see this when you inspect the page in your browser:


localhost:8080/chapter-1/getting-st
localhost:8080/chapter-1/getting-started.html
60 FPS (58-60)
canvas 1267 × 677
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Search HTML
Filter Styles
:hov .cls
Layout
Computed
Changes
Fonts
Anima
Flexbox
Grid
Box Model
margin
border
padding
0
0
0
html › body › canvas

```
<!DOCTYPE html>
<html> event scroll
 <head>…</head>
 <body>
   <canvas style="display: block; width: 1267px; height: 677px; touch-action: none;" data-engine="three.js
   r137" width="1267" height="677"  ></canvas> event overflow
  <div style="position: fixed; top: 0px; left: 0px; cursor: pointer; opacity: 0.9; z-index: 10000;">…
   </div> event overflow
 </body>
</html>
```

```
element {                inline
   display: block;
   width: 1267px;
   height: 677px;
   touch-action: none;
}
```

Select a Flex container or item to continue.

CSS Grid is not in use on this page

Figure 1.9 – Canvas added by Three.js

At this point, we've got an empty `THREE.Scene`, a `THREE.PerspectiveCamera`, and a `THREE.WebGLRenderer`. If we add some objects to the scene, we can already show some output on the screen. Before we do this, though, we'll add a couple of additional components:

- **OrbitControls**: This will allow you to use the mouse to rotate and pan around the scene
- **Lights**: This allows us to use some more advanced materials, cast shadows, and generally make our scene look better

In the next section, we'll first add the lights.

Adding lights

If we don't have lights in the scene, most materials will be rendered in black. So, to see our meshes (and get shadows), we're going to add some lights to the scene. In this case, we're going to add two lights:

- `THREE.AmbientLight`: This is just a simple light that affects everything with the same intensity and color.
- `THREE.DirectionalLight`: This is a light source whose rays are cast in parallel to one another. This is pretty much how we experience the light of the Sun.

The following code fragment shows how to do this:

```
// add lights
scene.add(new THREE.AmbientLight(0x666666))
const dirLight = new THREE.DirectionalLight(0xaaaaaa)
dirLight.position.set(5, 12, 8)
dirLight.castShadow = true
// and some more shadow related properties
```

And once again, these lights can be configured in various ways, the details of which we'll explain in *Chapter 3*. At this point, we've got all the components ready to render a scene, so let's add the meshes.

Adding the meshes

In the following code fragment, we create the three meshes in our scene:

```
// create a cube and torus knot and add them to the scene
const cubeGeometry = new THREE.BoxGeometry();
const cubeMaterial = new THREE.MeshPhongMaterial({ color:
  0x0000FF });
const cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
cube.position.x = -1;
cube.castShadow = true;
scene.add(cube);
const torusKnotGeometry = new THREE.
TorusKnotBufferGeometry(0.5, 0.2, 100, 100);
const torusKnotMat = new THREE.MeshStandardMaterial({
  color: 0x00ff88,
  roughness: 0.1,
});
const torusKnotMesh = new THREE.Mesh(torusKnotGeometry,
torusKnotMat);
torusKnotMesh.castShadow = true;
torusKnotMesh.position.x = 2;
scene.add(torusKnotMesh);
// create a very large ground plane
const groundGeometry = new THREE.PlaneBufferGeometry(10000,
  10000)
```

```
const groundMaterial = new THREE.MeshLambertMaterial({
  color: 0xffffff
})
const groundMesh = new THREE.Mesh(groundGeometry,
groundMaterial)
groundMesh.position.set(0, -2, 0)
groundMesh.rotation.set(Math.PI / -2, 0, 0)
groundMesh.receiveShadow = true
scene.add(groundMesh)
);
```

Here, we have created a cube, a torus knot, and the ground. All these meshes follow the same idea:

1. We create the shape – that is, the geometry of the objects: a `THREE.BoxGeometry`, a `THREE.TorusKnotBufferGeometry`, and a `THREE.PlaneBufferGeometry`.
2. We create the material. In this case, we use a `THREE.MeshPhongMaterial` for the cube, a `THREE.MeshStandardMaterial` for the torus knot, and a `THREE.MeshLambertMaterial` for the ground. The color of the cube is blue, the color of the torus knot is greenish, and the color of the ground is white. In *Chapter 4*, *Working with Three.js Materials*, we're going to explore all these materials, where they can best be used, and how to configure them.
3. We tell Three.js that the cube and the torus knot cast shadows and that the ground will receive shadows.
4. Finally, from the shape and the material, we create a `THREE.Mesh`, position the mesh, and add it to the scene.

At this point, we just have to call `renderer.render(scene, camera)`. You will see the result on your screen:

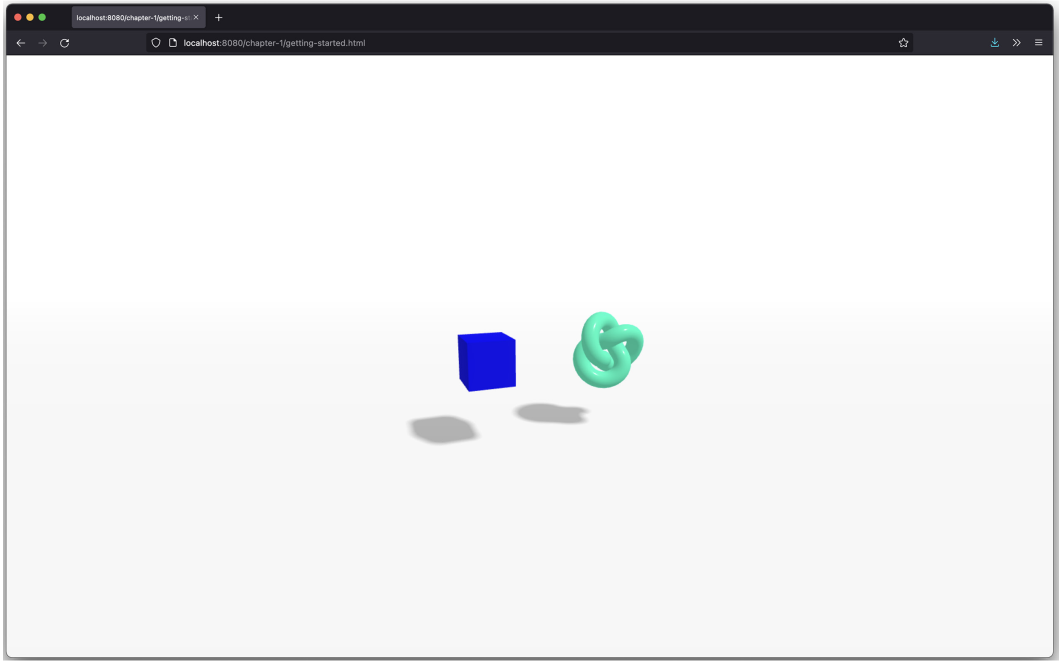

localhost:8080/chapter-1/getting-st
localhost:8080/chapter-1/getting-started.html

Figure 1.10 – Geometries renderer – static

If you've got the source file (`chapter-01/getting-started.js`), open it in your editor; now is also a good time to experiment a bit with the settings. By changing the `torusKnot.postion.x`, `torusKnot.position.y`, and `torusKnot.position.z` settings, you can move the torus knot around the scene (changes are applied after you save the file in the editor). You can also easily change the color of the meshes by changing the `color` property of the materials.

Adding an animation loop

At this point, the scene is very static. You can't move the camera around, and nothing is moving. If we want to animate the scene, the first thing that we need to do is find some way to re-render the scene at a specific interval. Before HTML5 and the related JavaScript APIs came along, the way to do this was by using the `setInterval(function,interval)` function. With `setInterval`, we could specify a function that, for instance, would be called every 100 milliseconds. The problem with this function is that it doesn't take into account what is happening in the browser. If you were browsing another tab, this function would still be fired every couple of milliseconds. Besides that, `setInterval` isn't synchronized when the the screen is redrawn. This can lead to higher CPU usage, flickering, and generally poor performance.

Luckily, modern browsers have a solution for that with the `requestAnimationFrame` function.

Introducing requestAnimationFrame

With `requestAnimationFrame`, you can specify a function that is called at an interval. However, you don't define this interval. This interval is defined by the browser. You do any drawing you need to do in the supplied function, and the browser will make sure it is painted as smoothly and efficiently as possible. Using this is simple. We just add the following code:

```
function animate() {
  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();
```

In the preceding `animate` function, we called `requestAnimationFrame` again, to keep the animation going. The only thing we need to change in the code is that instead of calling `renderer.render` after we've created the complete scene, we call the `animate()` function once to initiate the animation. If you run this, you won't see any changes yet compared to the previous example because we haven't changed anything in this `animate()` function. Before we add additional functionality to this function though, we will introduce a small helper library called **stats.js**, which gives us information about the frame rate the animation is running at. This library, from the same author as Three.js, renders a small graph that shows us information about the rate at which the scene is rendered.

To add these statistics, all we need to do is import the correct module and add it to our page:

```
import Stats from 'three/examples/jsm/libs/stats.module'
const stats = Stats()
document.body.appendChild(stats.dom)
```

If you leave it at this, you'll see a nice stats counter in the top left of your screen, but nothing will happen. The reason is that we need to tell this element when we're in the `requestAnimationFrame` loop. For this, we just need to add the following to our `animate` function:

```
function animate() {
  requestAnimationFrame(animate);
  stats.update();
  renderer.render(scene, camera);
}
animate();
```

If you open the `chapter-1/getting-started.html` example, you'll see that it shows a **frames per second** (**FPS**) counter in the top left of your screen:

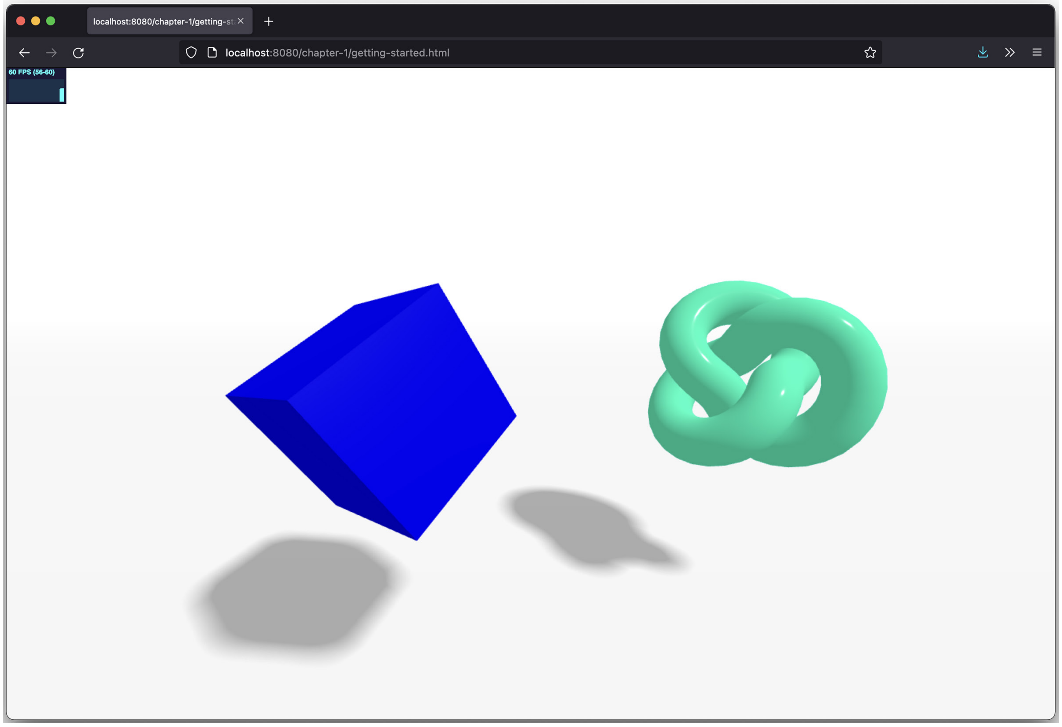

localhost:8080/chapter-1/getting-st
localhost:8080/chapter-1/getting-started.html
60 FPS (56-60)

Figure 1.11 – FPS statistics

In the `chapter-1/getting-started.html` example, you can already see that the torus knot and cube are moving around their axes. In the following section, we'll explain how you do this by extending the `animate()` function.

Animating the meshes

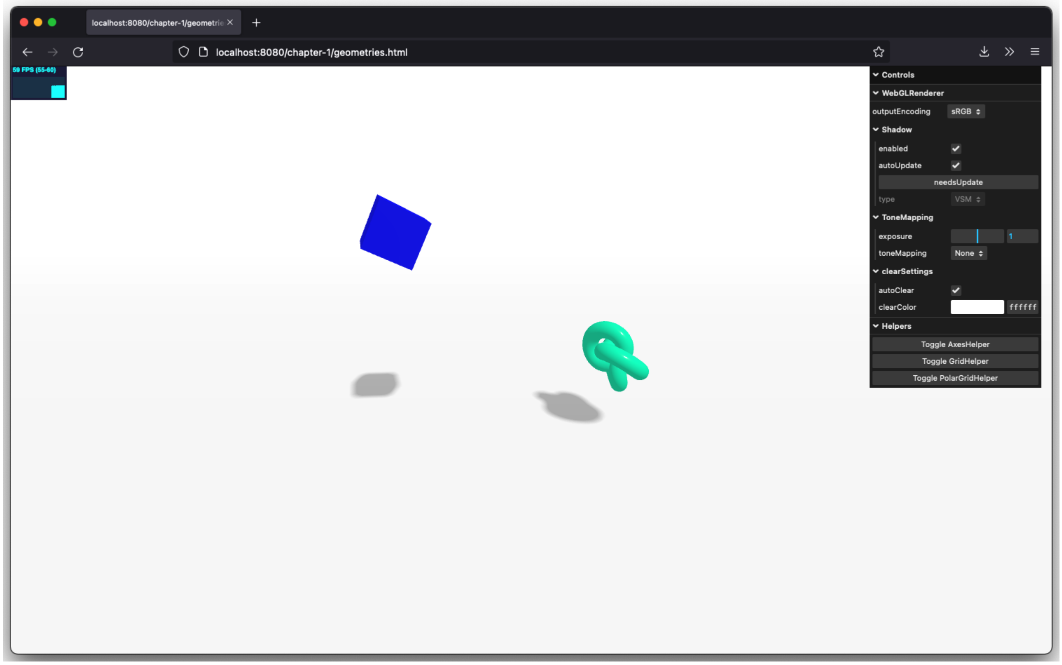
With `requestAnimationFrame` and the statistics configured, we've got a place to put our animation code. All we need to do is add this to the `animate()` function:

```
cube.rotation.x += 0.01;
cube.rotation.y += 0.01;
cube.rotation.z += 0.01;
torusKnotMesh.rotation.x -= 0.01;
torusKnotMesh.rotation.y += 0.01;
torusKnotMesh.rotation.z -= 0.01;
```

That looks simple, right? What we do is increase the rotation property of each of the axes by 0.01 every time the `animate()` function is called, which shows up as the meshes smoothly rotating around all of their axes. If we change the position instead of the rotation around the axes, we can move the meshes around:

```
let step = 0;
animate() {
  ...
  step += 0.04;
  cube.position.x = 4*(Math.cos(step));
  cube.position.y = 4*Math.abs(Math.sin(step));
  ...
}
```

With the cube, we have already changed the `rotation` property; now, we're also going to change its `position` property in the scene. We want the cube to bounce from one point in the scene to another with a nice, smooth curve. For this, we need to change its position on the *x*-axis, as well as the *y*-axis. The `Math.cos` and `Math.sin` functions help us create a smooth trajectory using the `step` variable. I won't go into the details of how this works here. For now, all you need to know is that `step+=0.04` defines the speed of the bouncing sphere. If you want to enable this for yourself, open up the `chapter-1/geometries.js` file and uncomment the section from the `animate()` function. Once you've done this, you'll see something like this on screen, where the blue cube is dancing around the scene:


localhost:8080/chapter-1/geometrie
localhost:8080/chapter-1/geometries.html
59 FPS (55-60)
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

outputEncoding sRGB
enabled ☑
autoUpdate ☑
type VSM
exposure 1
toneMapping None
autoClear ☑
clearColor ffffff

Figure 1.12 – Jumping blue cube

Enabling orbit controls

If you try and move the scene around with your mouse, nothing much will happen. That is because we added the camera to a fixed position, and we didn't update its position in the animate loop. We can, of course, do this in the same manner as we did to the position of the cube, but Three.js comes with several *controls* that allow you to easily pan around the scene and move the camera around. For this example, we'll introduce `THREE.OrbitControls`. With these controls, you can use your mouse to move the camera around the scene and look at different objects. All we need to do to get this working is create a new instance of these controls, attach them to the camera, and call the `update` function from our animation loop:

```
const orbitControls = new OrbitControls(camera, renderer.
  domElement)
// and the controller has a whole range of other properties we
can set
function animate() {
  ...
  orbitControls.update();
}
```

Now, you can use your mouse to navigate around the scene. This is already enabled in the `chapter-1/getting-started.html` example:

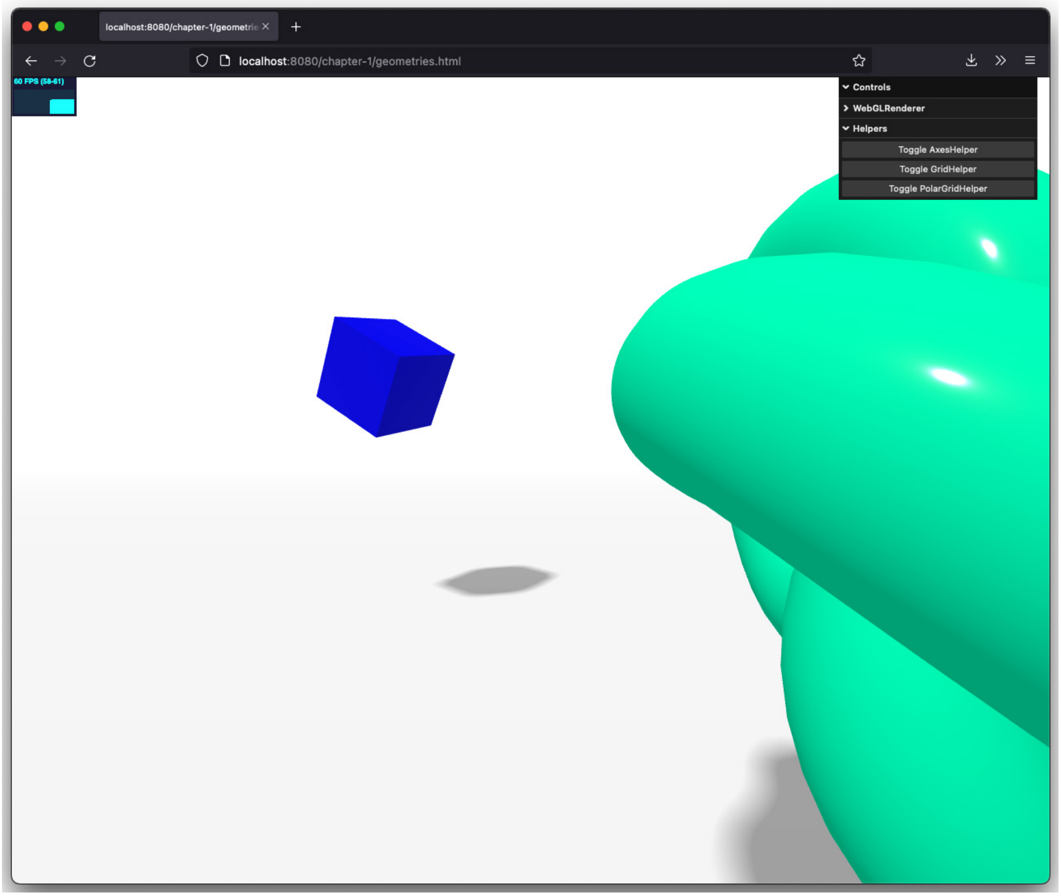

localhost:8080/chapter-1/geometrie
localhost:8080/chapter-1/geometries.html
60 FPS (58-61)
Controls
WebGLRenderer
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

Figure 1.13 – Zooming in with orbit controls

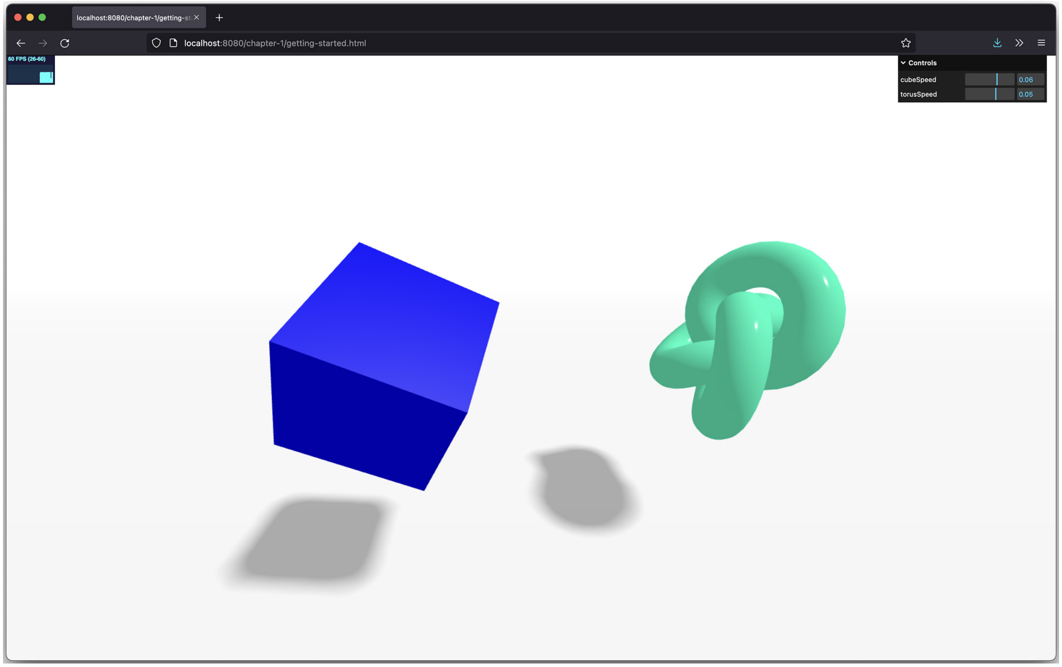
Before wrapping up this section, we will add one more element to our basic scene. When working with 3D scenes, animations, colors, and properties, it often requires a bit of experimenting to get the correct color, animation speed, or material property. It would be very easy if you had a simple **GUI** that allowed you to change these kinds of properties on the fly. Luckily, you do!

Using lil-gui to control properties and make experimenting easier

In the previous example, we added a little bit of animation for the torus knot and the cube. Now, we'll create a simple UI element that allows us to control the speed of the rotations and the movement. For this, we're going to use the `lil-gui` library from `https://lil-gui.georgealways.com/`. This library allows us to quickly create a simple control UI to make experimenting with the scene easier. It can be added as follows:

```
import GUI from "lil-gui";
...
const gui = new GUI();
const props = {
  cubeSpeed: 0.01,
  torusSpeed: 0.01,
};
gui.add(props, 'cubeSpeed', -0.2, 0.2, 0.01)
gui.add(props, 'torusSpeed', -0.2, 0.2, 0.01)
function animate() {
  ...
  cube.rotation.x += props.cubeSpeed;
  cube.rotation.y += props.cubeSpeed;
  cube.rotation.z += props.cubeSpeed;
  torusKnotMesh.rotation.x -= props.torusSpeed;
  torusKnotMesh.rotation.y += props.torusSpeed;
  torusKnotMesh.rotation.z -= props.torusSpeed;
  ...
}
```

In the preceding code fragment, we created a new control element (`new GUI`) and configured two controls: `cubeSpeed` and `torusSpeed`. In each animation step, we'll just look up the current values and use those to rotate the meshes. Now, we can experiment with the properties without having to switch between the browser and the editor. You'll see this UI in most of the examples in this book where we provide it so that you can easily play around with the different options provided by the materials, the lights, and the other Three.js objects. In the following screenshot, you can see the controls you can use to control the scene in the top-right part of the screen:


localhost:8080/chapter-1/getting-st
localhost:8080/chapter-1/getting-started.html
60 FPS (26-60)
Controls

cubeSpeed 0.06
torusSpeed 0.05

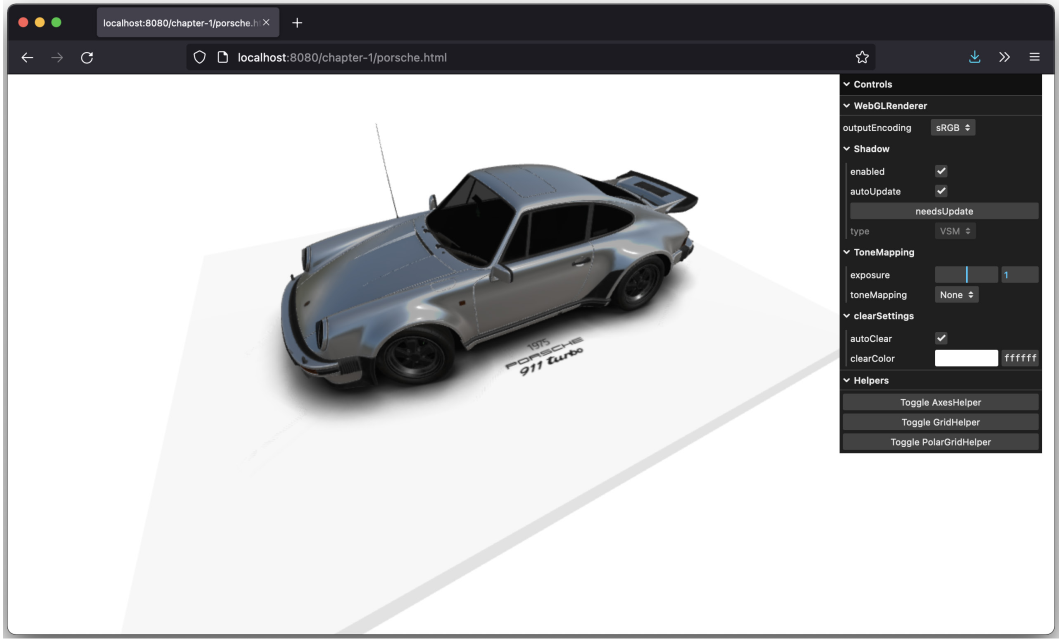
Figure 1.14 – Using controls to modify the properties of the scene

Before we move on to the last section of this chapter, here's a quick note on what we've shown so far. You can imagine that most scenes will need pretty much the same setup. They all need some lights, a camera, a scene, and maybe a ground floor. To avoid having to add all this to each example, we've externalized most of these common elements to a set of helper libraries. That way, we can keep the examples nice and clean so that they only show you the code relevant to that example. If you're interested in how that's set up, you can look at the files from the `bootstrap` folder, which brings this approach together.

In the previous example, we rendered some simple meshes in the scene and positioned them directly. Sometimes, though, it is hard to determine where to position objects, or how far we should rotate them. Three.js provides several different helpers that provide you with additional information about the scene. In the next section, we'll look at a couple of these helper functions.

Helper objects and util functions

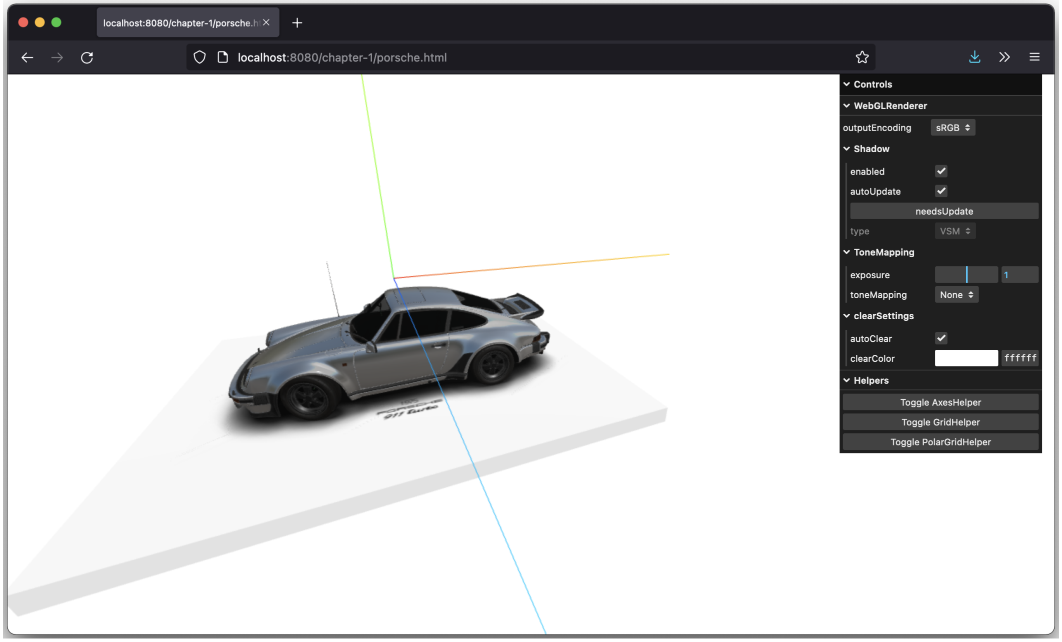
Before we move on to the next chapter, we're going to quickly introduce a couple of helper functions and objects. These helpers make it easier to position objects and see what is happening in a scene. The easiest way to see this in action is to open the `chapter-01/porsche.html` example in your browser:


localhost:8080/chapter-1/porsche.h
localhost:8080/chapter-1/porsche.html
1975
PORSCHE
911 turbo
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

outputEncoding sRGB
enabled ☑
autoUpdate ☑
type VSM
exposure 1
toneMapping None
autoClear ☑
clearColor ffffff

Figure 1.15 – Porsche example with helper

On the right-hand side of the screen, at the bottom of the menu, you will see three buttons in the controls: **Toggle AxesHelper**, **Toggle GridHelper**, and **Toggle PolarGridHelper**. When you click on any of them, Three.js will add an overlay to the screen that can help you orient and position meshes, determine needed rotations, and check the sizes of your objects. For instance, when we toggle **AxesHelper**, we will see the *x*-, *y*-, and *z*-axes in the scene:


localhost:8080/chapter-1/porsche.h
localhost:8080/chapter-1/porsche.html
Controls
WebGLRenderer
Shadow
needsUpdate
ToneMapping
clearSettings
Helpers
Toggle AxesHelper
Toggle GridHelper
Toggle PolarGridHelper

outputEncoding sRGB
enabled ☑
autoUpdate ☑
type VSM
exposure 1
toneMapping None
autoClear ☑
clearColor ffffff

Figure 1.16 – Porsche example with AxesHelper enabled

Note that in this example, you can see a more extensive control UI, where you can also control various aspects of `WebGLRenderer`.

Summary

That's it for this first chapter. In this chapter, you learned how to set up your development environment, how to get the code, and how to get started with the examples provided in this book. Then, you learned that to render a scene with Three.js, you have to create a `THREE.Scene` object and add a camera, a light source, and the objects that you want to render. We also showed you how you can expand this basic scene by adding animations. Lastly, we added a couple of helper libraries. We used *lil-GUI*, which allows you to quickly create control UIs, and we added an FPS counter, which provided feedback on the frame rate and other metrics using which your scene is rendered.

All these items will help you understand the examples in upcoming chapters and make it easier for you to experiment with the more advanced examples and start modifying those to your liking. Should stuff break or not result in what you expect in the next few chapters when you experiment, remember what we showed you in this chapter: use the JavaScript console to get additional information, add debug statements, use the helpers provided by Three.js, or add custom control elements.

In the next chapter, we'll expand on the basic setup shown here and you'll learn more about the most important building blocks that you can use in Three.js.

2

The Basic Components that Make up a Three.js Application

In the previous chapter, you learned about the basics of Three.js. We looked at a couple of examples, and you created your first complete Three.js application. In this chapter, we'll dive a bit deeper into Three.js and explain the basic components that make up a Three.js application.

By the end of this chapter, you'll have learned how to use the basic components that are used in every Three.js application and should be able to create simple scenes using these standard components. You should also feel comfortable working with Three.js applications that use the more advanced objects, since the approach used by Three.js for simple and advanced components is the same.

In this chapter, we will cover the following topics:

- Creating a scene
- How geometries and meshes are related
- Using different cameras for different scenes

We'll start by looking at how you can create a scene and add objects.

Creating a scene

In *Chapter 1*, *Creating Your First 3D Scene with Three.js*, you created a `THREE.Scene`, so you already know some of the basics of Three.js. We saw that for a scene to show anything, we need four different types of objects:

- **Camera**: This determines which part of `THREE.Scene` is rendered onscreen.
- **Lights**: These have an effect on how materials are shown and are used when creating shadow effects (discussed in detail in *Chapter 3*, *Working with Light Sources in Three.js*).

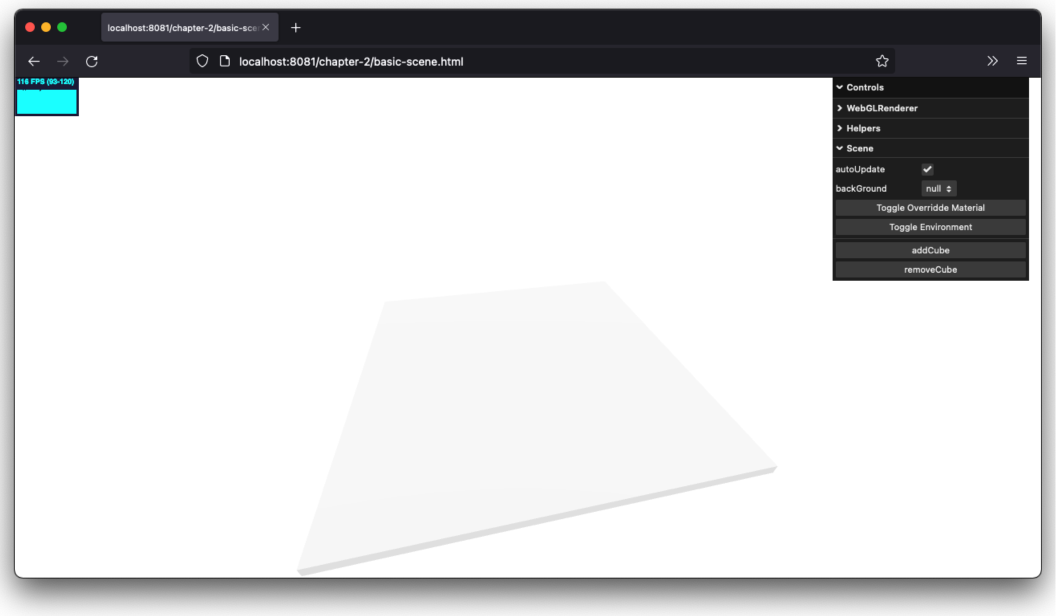
- **Meshes**: These are the main objects that are rendered from the perspective of the camera. These objects contain the vertices and faces that make up the geometry (for example, a sphere or a cube) and contain a material, which defines what the geometry looks like.
- **Renderer**: This uses the camera and the information in the scene to draw (render) the output on the screen.

`THREE.Scene` serves as the main container for the lights and the meshes you want to render. `THREE.Scene` itself doesn't have that many options and functions.

`THREE.Scene` is a structure that is sometimes also called a scene graph. A scene graph can hold all the necessary information of a graphical scene. In Three.js, this means that a `THREE.Scene` contains all the objects necessary for rendering. It is interesting to note that a scene graph, as the name implies, isn't just an array of objects; a scene graph consists of a set of nodes in a tree structure. As we'll see in *Chapter 8*, *Creating and Loading Advanced Meshes and Geometries*, Three.js provides objects you can use to create groups of different meshes or lights. The main object you use for that, which you can use to create a scene graph, is the `THREE.Group`. As the name implies, this object allows you to group objects together. A `THREE.Group` extends from another base class in Three.js called `THREE.Object3D`, which provides a set of standard functions to add and modify children. `THREE.Mesh` and `THREE.Scene` both also extend from a `THREE.Object3D` so you could also use those to create a nested structure. But it's convention, and also more semantically correct, to use `THREE.Group` to build up the scene graph.

The basic functionality of a scene

The best way to explore the functionality of a scene is by looking at an example. In the source code for this chapter, you can find the `chapter-2/basic-scene.html` example. We'll use this example to explain the various functions and options a scene has. When we open this example in the browser, the output will look similar to what's shown in the next screenshot (remember that you can use the mouse to move, zoom, and pan around the rendered scene):


localhost:8081/chapter-2/basic-sce
localhost:8081/chapter-2/basic-scene.html
116 FPS (93-120)
Controls
WebGLRenderer
Helpers
Scene
Toggle Overridde Material
Toggle Environment
addCube
removeCube

autoUpdate ☑
backGround null

Figure 2.1 – Basic scene setup

The preceding figure looks like the examples we saw in *Chapter 1*, *Creating Your First 3D Scene with Three.js*. Even though the scene looks pretty empty, it already contains a few objects:

- We've got `THREE.Mesh`, which represents the floor area you can see
- We're using `THREE.PerspectiveCamera` to determine what we're looking at
- We've added `THREE.AmbientLight` and `THREE.DirectionalLight` to provide lighting

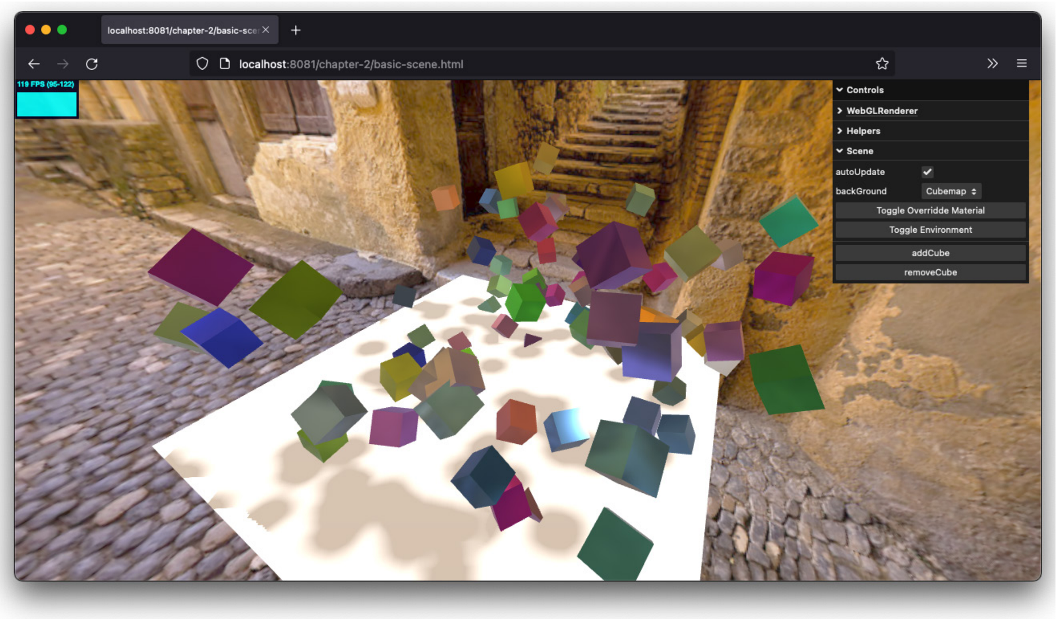
The source code for this example can be found in `basic-scene.js`, and we can use code from `bootstrap/bootstrap.js`, `bootstrap/floor.js`, and `bootstrap/lighting.js` since this is a generic scene setup we use throughout this book. What happens in all these files can be simplified as in the following code:

```
// create a camera
const camera = new THREE.PerspectiveCamera(
  75,
  window.innerWidth / window.innerHeight,
  0.1,
  1000
);
// create a renderer
```

```
const renderer = new THREE.WebGLRenderer({ antialias: true
  });
// create a scene
const scene = new THREE.Scene();
// create the lights
scene.add(new THREE.AmbientLight(0x666666));
scene.add(THREE.DirectionalLight(0xaaaaaa));
// create the floor
const geo = new THREE.BoxBufferGeometry(10, 0.25, 10, 10,
  10, 10);
const mat = new THREE.MeshStandardMaterial({ color:
  0xffffff,});
const mesh = new THREE.Mesh(geo, mat);
scene.add(mesh);
```

As you can see in the preceding code, we create `THREE.WebGLRenderer` and `THREE.PerspectiveCamera`, since we always need those. Next, we create a `THREE.Scene` and just add all the objects that we want to use. In this case, we add two lights and a single mesh. Now, we have all the components to start up a render loop, as we've already seen in *Chapter 1*, *Creating Your First 3D Scene with Three.js*.

Before we look at the `THREE.Scene` object in more depth, we'll first explain what you can do in the demo, and after that, look at the code. Open the `chapter-2/basic-scene.html` example in your browser and look at the **Controls** menu in the upper-right corner, which you can see in the following screenshot:



localhost:8081/chapter-2/basic-sce
localhost:8081/chapter-2/basic-scene.html
119 FPS (95-122)
Controls
WebGLRenderer
Helpers
Scene
Toggle Overridde Material
Toggle Environment
addCube
removeCube

autoUpdate ☑

backGround Cubemap

Figure 2.2 – Basic scene setup with Cubemap background

Adding and removing objects

With these **Controls**, you can add cubes to the scene and remove the cube that was last added. It also allows you to change the background of the scene and set the material and the environment map for all the objects in the scene. We'll explore these various options and how you can use them to configure a `THREE.Scene`. We'll start by looking at how you can add and remove `THREE.Mesh` objects to and from a scene. The following code shows the function we call when you click on the **addCube** button:

```
const addCube = (scene) => {
  const color = randomColor();
  const pos = randomVector({
    xRange: { fromX: -4, toX: 4 },
    yRange: { fromY: -3, toY: 3 },
    zRange: { fromZ: -4, toZ: 4 },
  });
  const rotation = randomVector({
    xRange: { fromX: 0, toX: Math.PI * 2 },
    yRange: { fromY: 0, toY: Math.PI * 2 },
    zRange: { fromZ: 0, toZ: Math.PI * 2 },
  });
  const geometry = new THREE.BoxGeometry(0.5, 0.5, 0.5);
  const cubeMaterial = new THREE.MeshStandardMaterial({
    color: color,
    roughness: 0.1,
    metalness: 0.9,
  });
  const cube = new THREE.Mesh(geometry, cubeMaterial);
  cube.position.copy(pos);
  cube.rotation.setFromVector3(rotation);
  cube.castShadow = true;
  scene.add(cube);
};
```

Let us understand the preceding code in detail:

- First, we have determined some random settings for the cube that will be added: a random color (by calling the `randomColor()` helper function), a random position, and a random rotation. These last two are randomly generated by calling `randomVector()`.
- Next, we create the geometry we want to add to the scene: a cube. We just create a new `THREE.BoxGeometry` for this, define a material (`THREE.MeshStandardMaterial` in this example), and combine these two into `THREE.Mesh`. We use random variables to set the cube's position and rotation.
- Finally, this `THREE.Mesh` can then be added to the scene by calling `scene.add(cube)`.

A new element that we have introduced in the preceding code is that we also give the cube a name using the `name` attribute. The name is set to `cube-`, appended with the number of objects currently in the scene (`scene.children.length`). A name is very useful for debugging purposes but can also be used to directly access an object from your scene. If you use the `THREE.Scene.getObjectByName(name)` function, you can directly retrieve a specific object and, for instance, change its location without having to make the JavaScript object a global variable.

There might also be situations where you want to remove an existing object from a `THREE.Scene`. Since a `THREE.Scene` exposes all its children through the `children` property, we can just use the following simple code to remove the last child added:

```
const removeCube = (scene) => {
  scene.children.pop();
};
```

Three.js provides other helpful functions for the `THREE.Scene` too, related to working with the children of the scene:

- `add`: We've already seen this function, which adds the provided object to the scene. If it was previously added to a different `THREE.Object3D`, it'll be removed from that object.
- `Attach`: This is similar to `add`, but if you use it, any rotations or translations applied to this object will be kept.
- `getObjectById`: When you add an object to a scene, it gets an ID. The first one gets `1`, the second one `2`, and so on. With this function, you can get a child based on this ID.
- `getObjectByName`: This returns an object based on its `name` property. The name is something you can set on an object – this is in contrast with the `id` property, which is assigned by Three.js.
- `Remove`: This removes this object from the scene.
- `Clear`: This removes all the children from the scene.

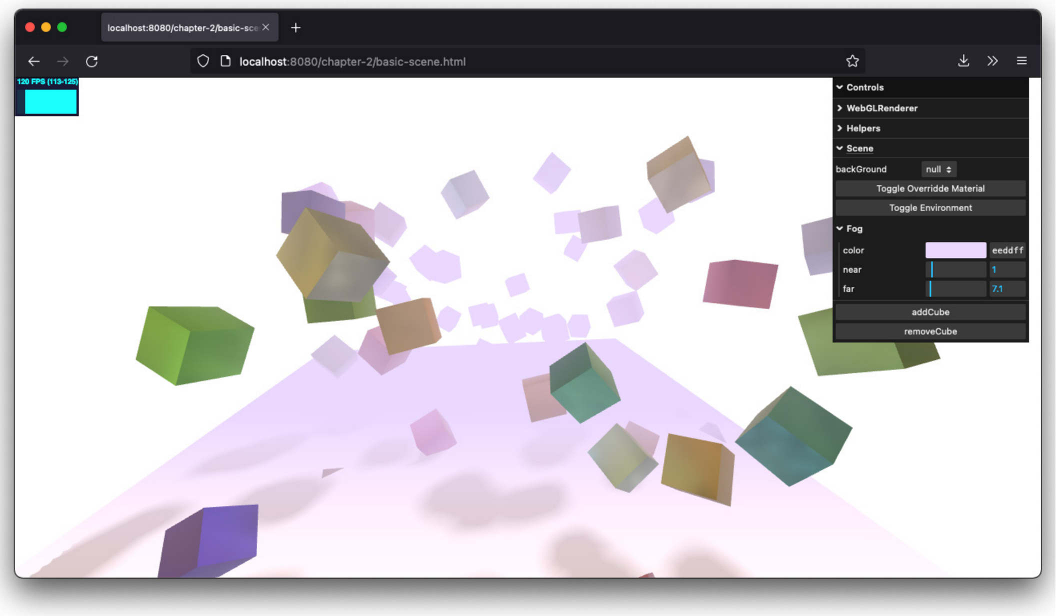
Note that the preceding functions are actually from the base object that the `THREE.Scene` extends from: the `THREE.Object3D`.

Throughout the book, we'll use these functions if we want to manipulate the children of a scene (or in `THREE.Group`, as we'll explore later on.)

Besides the functionality to add and remove objects, a `THREE.Scene` also provides a couple of other settings. The first one we'll look at is adding fog.

Adding fog

The `fog` property lets you add a fog effect to the complete scene; the farther an object is from the camera, the more it will be hidden from sight. This is shown in the following screenshot:


localhost:8080/chapter-2/basic-sce
localhost:8080/chapter-2/basic-scene.html
120 FPS (113-125)
Controls
WebGLRenderer
Helpers
Scene
Toggle Overridde Material
Toggle Environment
Fog
addCube
removeCube

backGround null

color eeddff

near 1

far 7.1

Figure 2.3 – Using fog to hide objects

To best see the effect of the added fog, use the mouse to zoom in and out, and you'll see the cubes being affected by the fog. Enabling fog is really easy in Three.js. Just add the following line of code after you've defined your scene:

```
scene.fog = new THREE.Fog( 0xffffff, 1, 20 );
```

Here, we define white fog (`0xffffff`). The other two properties can be used to tune how the mist appears. The `1` value sets the `near` property, and the `20` value sets the `far` property. With these properties, you can determine where the mist starts and how fast it gets denser. With the `THREE.Fog` object, the fog increases linearly. In the `chapter-02/basic-scene.html` sample, you can modify these properties by using the menu on the right of the screen to see how these settings affect what you see onscreen.

There is also an alternative fog implementation provided by Three.js, `THREE.FogExp2`:

```
scene.fog = new THREE.FogExp2( 0xffffff, 0.01 );
```

This time, we don't specify near and far, but just the color (`0xffffff`) and the mist's density (`0.01`). Usually, it's best to experiment a bit with these properties to get the effect you want.

Another interesting feature of a scene is that you can configure a background.

Changing the background

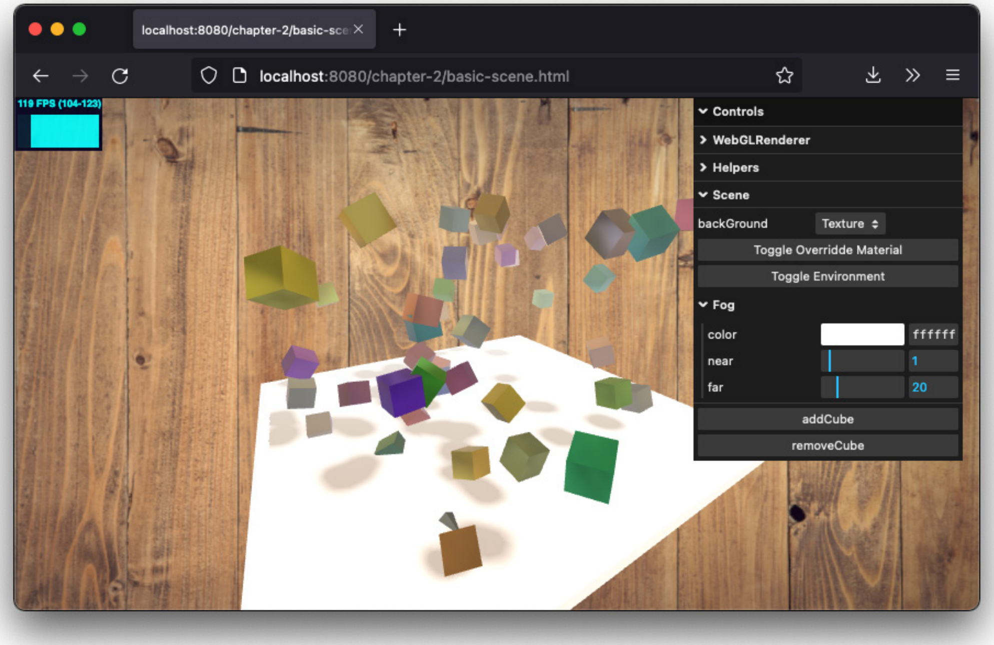
We've already seen that we can change the background color by setting `clearColor` of `WebGLRenderer` like this: `renderer.setClearColor(backgroundColor)`. You can also use the `THREE.Scene` object to change the background. For this, you've got three options:

- *Option 1*: You can use a solid color.
- *Option 2*: You can use a texture, which is basically an image, stretched out to fill the complete screen. (More on textures in *Chapter 10*, *Loading and Working with Textures*.)
- *Option 3*: You can use an environment map. This is also a kind of texture, but one that completely encompasses the camera, and moves around when you change the camera orientation.

Note that this sets the background color of the HTML canvas we're rendering to and not the background color of the HTML page. If you want to have a transparent canvas, you need to set the `alpha` property of the renderer to `true`:

```
new THREE.WebGLRenderer({ alpha: true })
```

In the `chapter-02/basic-scene.html` menu on the right, there is a dropdown that shows all these different settings. If you select the **Texture** option from the **backGround** dropdown, you'll see the following:


localhost:8080/chapter-2/basic-sce
localhost:8080/chapter-2/basic-scene.html
119 FPS (104-123)
Controls
WebGLRenderer
Helpers
Scene
Toggle Overridde Material
Toggle Environment
Fog
addCube
removeCube

backGround Texture

color ffffff

near 1

far 20

Figure 2.4 – Background using a texture

We'll cover textures and cubemaps in much more detail in *Chapter 10*, *Loading and Working with Textures*. But we'll have a quick look now at how to configure these and a simple background color for the scene (the source for this can be found in `controls/scene-controls.js`):

```
// remove any background by setting the background to null
scene.background = null;
// if you want a simple color, just set the background to a
  color
scene.background = new THREE.Color(0x44ff44);
// a texture can be loaded with a THREE.TextureLoader
const textureLoader = new THREE.TextureLoader();
textureLoader.load(
  "/assets/textures/wood/abstract-antique-
    backdrop-164005.jpg",
  (loaded) => {
    scene.background = loaded;
  }
// a cubemap can also be loaded with a THREE.TextureLoader
     textureLoader.load("/assets/equi.jpeg", (loaded) => {
  loaded.mapping = THREE.EquirectangularReflectionMapping;
  scene.background = loaded;
});
```

As you can see from the preceding code, you can assign either `null`, `THREE.Color`, or `THREE.Texture` to the `background` property of the scene. Loading a texture or a cubemap is done asynchronously, so, we have to wait for `THREE.TextureLoader` to load the image data before we can assign it to the background. In the case of the cubemap, we need to take an extra step and tell Three.js what kind of texture we've loaded. We will go into more detail in *Chapter 10*, *Loading and Working with Textures*, when we dive into the details of how textures work.

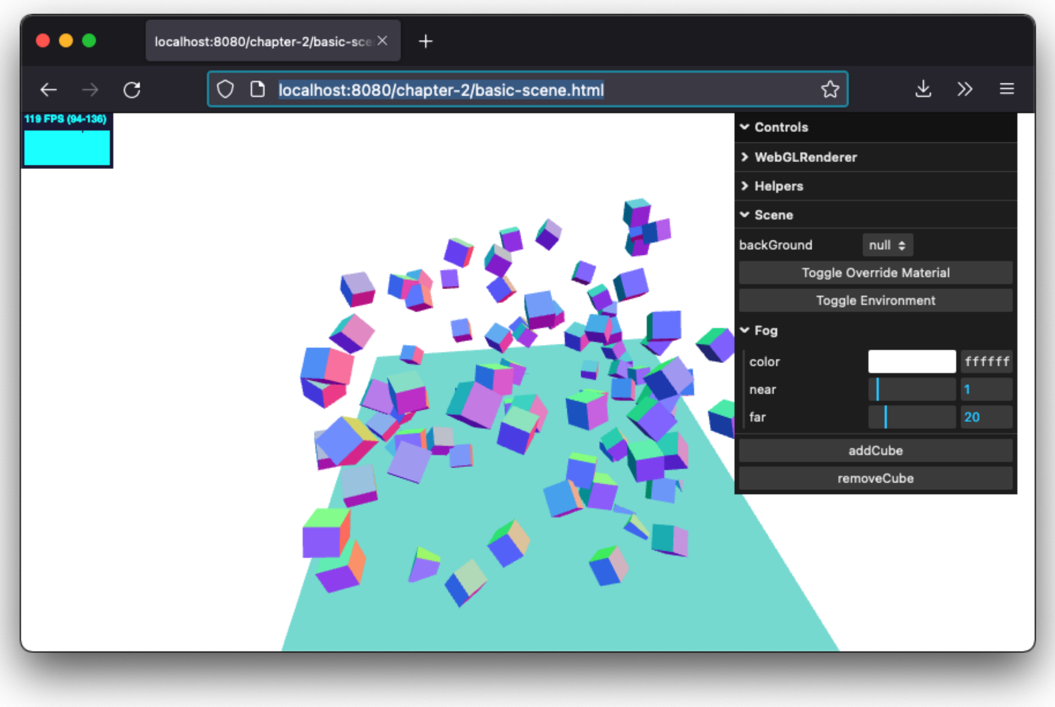
If you look back at the beginning of the following code section, you will see how we created the cubes that we added to the scene:

```
const geometry = new THREE.BoxGeometry(0.5, 0.5, 0.5);
const cubeMaterial = new THREE.MeshStandardMaterial({
  color: color,
  roughness: 0.1,
  metalness: 0.9,
});
const cube = new THREE.Mesh(geometry, cubeMaterial);
```

In the preceding code, we have created a geometry and specified a material. The `THREE.Scene` object also provides a way to force the meshes in the scene to use the same material. In the following section, we'll explore how that works.

Updating all the materials in the scene

A `THREE.Scene` has two properties that affect the material of the meshes in the scene. The first one is the `overrideMaterial` property. First, let's demonstrate how this works. On the `chapter-02/basic-scene.html` page, you can click on the **Toggle Override Material** button. This will change the material of all the meshes in the scene to a `THREE.MeshNormalNormal` material:



localhost:8080/chapter-2/basic-scene.html

119 FPS (94-136)

backGround: null

color: ffffff

near: 1

far: 20

localhost:8080/chapter-2/basic-sce
Controls
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
Fog
addCube
removeCube

Figure 2.5 – Override mesh materials with MeshNormalMaterial

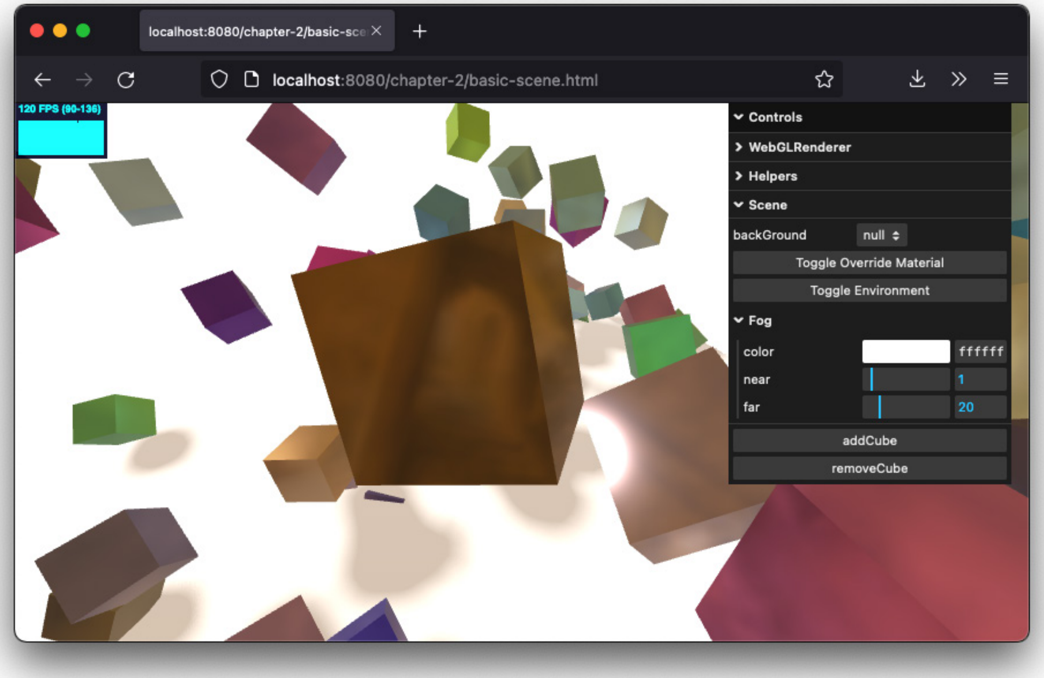
As you can see in the preceding figure, all the objects (including the ground floor) now use the same material – in this case, `THREE.MeshNormalMaterial`. This material colors each face of the mesh based on its orientation (its *normal* vector) to the camera. This can be very easily done in code by just calling `scene.overrideMaterial = new THREE.MeshNormalMaterial();`.

Besides applying a complete material to the scene, Three.js also provides a way to set the environment map property of each mesh's material to the same value. An environment map simulates the environment (for instance, a room, outdoors, or a cave) the meshes are in. The environment map can be used to create reflections on the meshes to make them feel more real.

We've already seen how we can load an environment map in the previous section on backgrounds. If we want all our materials to use an environment map for more dynamic reflections and shading, we can assign that loaded environment map to the `environment` property of a scene:

```
textureLoader.load("/assets/equi.jpeg", (loaded) => {
  loaded.mapping = THREE.EquirectangularReflectionMapping;
  scene.environment = loaded;
});
```

The best way to demonstrate the preceding code is by toggling the **Toggle Environment** button from the `chapter-02/basic-scene.html` example. If you now zoom in close to the cubes, you can see that their faces reflect part of the environment and aren't a solid color anymore:


localhost:8080/chapter-2/basic-sce
localhost:8080/chapter-2/basic-scene.html
120 FPS (90-136)
Controls
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
Fog
addCube
removeCube

backGround null

color ffffff

near 1

far 20

Figure 2.6 – Set the environment map to all the meshes in the scene

Now that we've discussed the basic container for all the objects we want to render, in the next section, we'll take a closer look at the objects (`THREE.Mesh` combining `THREE.Geometry` and a material) that you can add to the scene.

How geometries and meshes are related

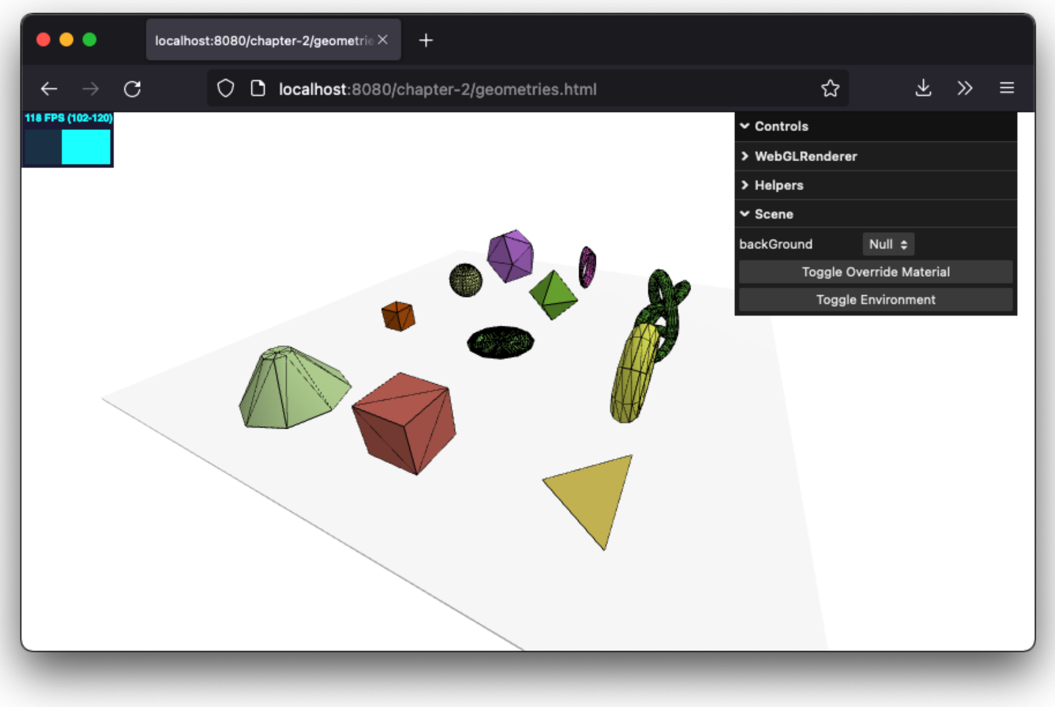
In each of the examples so far, you've seen geometries and meshes being used. For instance, to create a sphere and add it to the scene, we used the following:

```
const sphereGeometry = new THREE.SphereGeometry(4, 20, 20);
const sphereMaterial = new THREE.MeshBasicMaterial({color:
  0x7777ff);
const sphere = new THREE.Mesh(sphereGeometry,
  sphereMaterial);
scene.add(sphere);
```

We defined the geometry (`THREE.SphereGeometry`), which is the shape of an object, and its material (`THREE.MeshBasicMaterial`), and we combined these two in a mesh (`THREE.Mesh`) that can be added to a scene. In this section, we'll take a closer look at geometries and meshes. We'll start with geometries.

The properties and functions of a geometry

Three.js comes with a large set of geometries out of the box that you can use in your 3D scene. Just add a material, create a mesh, and you're pretty much done. The following screenshot, from the `chapter-2/geometries` example, shows a couple of the standard geometries available in Three.js:


localhost:8080/chapter-2/geometrie
localhost:8080/chapter-2/geometries.html
118 FPS (102-120)
Controls
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment

backGround Null

Figure 2.7 – Some of the basic geometries available in the scene

In *Chapter 5*, *Learning to Work with Geometries*, and *Chapter 6*, *Exploring Advanced Geometries*, we'll explore all the basic and advanced geometries that Three.js has to offer. For now, we'll look in greater detail at what a geometry actually is.

A geometry in Three.js, and in most other 3D libraries, is basically a collection of points in a 3D space, also called vertices (where a single point is called a vertex), and a number of faces connecting those points together. Take, for example, a cube:

- A cube has eight corners. Each of these corners can be defined as an `x-, y-, and z-` coordinate. So, each cube has eight points in a 3D space.
- A cube has six sides, with a vertex at each corner. In Three.js, a face always consists of three vertices that make a triangle (which has three edges). So, in the case of a cube, each side consists of two triangles to make the complete side. An example of how that looks can be seen in *Figure 2.7* by looking at the red cube.

When you use one of the geometries provided by Three.js, you don't have to define all the vertices and faces yourself. For a cube, you only need to define the width, height, and depth. Three.js uses that information and creates a geometry with eight vertices in the correct position and with the correct number of faces (12 in the case of a cube – 2 triangles per side). Even though you'd normally use the geometries provided by Three.js or generate them automatically, you can still create geometries completely by hand using vertices and faces, although this can quickly become complex, as you can see in the following lines of code:

```
const v = [
    [1, 3, 1],
    [1, 3, -1],
    [1, -1, 1],
    [1, -1, -1],
    [-1, 3, -1],
    [-1, 3, 1],
    [-1, -1, -1],
    [-1, -1, 1]]
const faces = new Float32Array([
  ...v[0], ...v[2], ...v[1],
  ...v[2], ...v[3], ...v[1],
  ...v[4], ...v[6], ...v[5],
  ...v[6], ...v[7], ...v[5],
  ...v[4], ...v[5], ...v[1],
  ...v[5], ...v[0], ...v[1],
  ...v[7], ...v[6], ...v[2],
  ...v[6], ...v[3], ...v[2],
  ...v[5], ...v[7], ...v[0],
  ...v[7], ...v[2], ...v[0],
  ...v[1], ...v[3], ...v[4],
  ...v[3], ...v[6], ...v[4]
]);
const bufferGeometry = new THREE.BufferGeometry();
bufferGeometry.setAttribute("position", new THREE.
BufferAttribute(faces, 3));
bufferGeometry.computeVertexNormals();
```

The preceding code shows how to create a simple cube. We define the points (the vertices) that make up this cube in the `v` array. From these vertices, we can create the faces next. In Three.js, we need to provide all the `faces` information in one large `Float32Array`. As we mentioned, a face consists of three vertices. So, for each face, we need to define nine values: the `x`, `y`, and `z` of each vertex. Since we've got three vertices per face, we have nine values. To make it a little bit easier to read, we use the

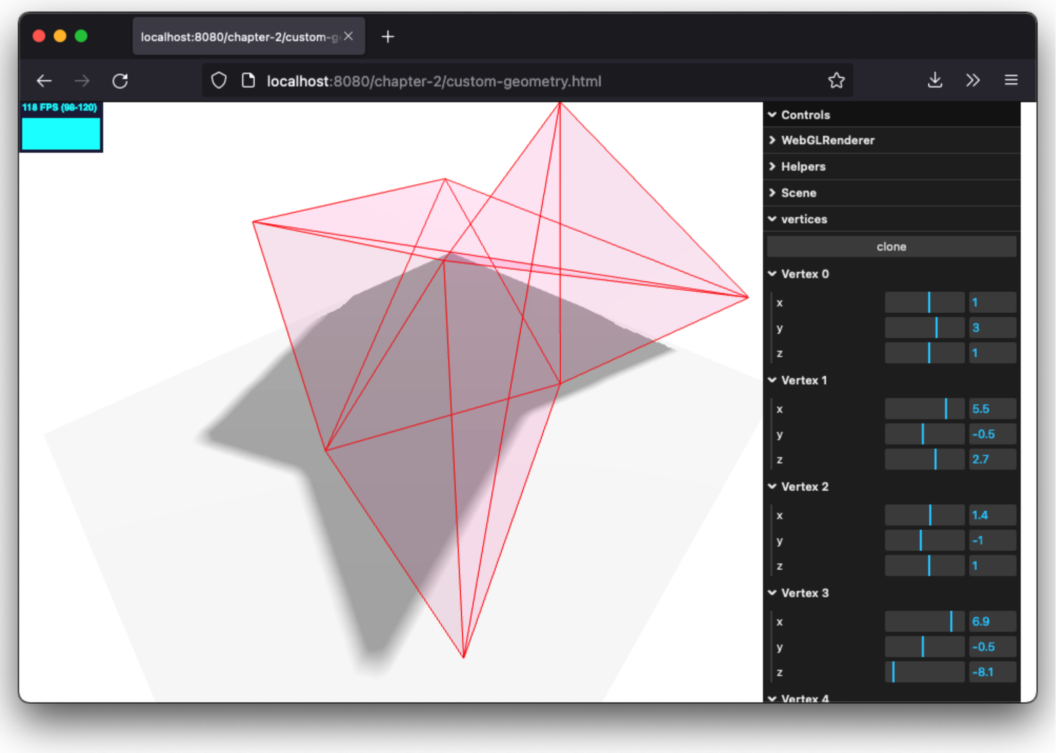
`...` (spread) operator from JavaScript to add the individual values of each vertex to the array. So, `...v[0], ...v[2], ...v[1]` will result in the following values in the array: `1, 3, 1, 1, -1, 1, 1, 3, 1`.

Note that you have to take care of the sequence of the vertices used to define the faces. The order in which they are defined determines whether Three.js thinks it is a front-facing face (a face facing the camera) or a back-facing face. If you create faces, you should use a clockwise sequence for front-facing faces and a counter-clockwise sequence if you want to create a back-facing face.

In our example, we have used a number of vertices to define the six sides of the cube, with two triangles for each face. In previous versions of Three.js, you could also use a quad instead of a triangle. A quad uses four vertices instead of three to define the face. Whether using quads or triangles is better is a heated debate raging in the 3D modeling world. Basically though, using quads is often preferred during modeling since they can be more easily enhanced and smoothed than triangles. For rendering and game engines though, working with triangles is often easier since every shape can be rendered very efficiently using triangles.

Using these vertices and faces, we can now create a new instance of `THREE.BufferGeometry` and assign the vertices to the `position` attribute. The last step is to call `computeVertexNormals()` on the geometry we have created. When we call this function, Three.js determines the normal vector for each of the vertices and for the faces. This is the information Three.js uses to determine how to color the faces based on the various lights in the scene (which you can easily visualize if you use `THREE.MeshNormalMaterial`).

With this geometry, we can now create a mesh, just like we saw earlier. We've created an example that you can use to play around with the position of the vertices, which also shows the individual faces. In our `chapter-2/custom-geometry` example, you can change the position of all the vertices of a cube and see how the faces react. This is shown in the following screenshot:


localhost:8080/chapter-2/custom-g
localhost:8080/chapter-2/custom-geometry.html
118 FPS (98-120)
Controls
WebGLRenderer
Helpers
Scene
vertices
clone
Vertex 4

Vertex 0
x 1
y 3
z 1
Vertex 1
x 5.5
y -0.5
z 2.7
Vertex 2
x 1.4
y -1
z 1
Vertex 3
x 6.9
y -0.5
z -8.1

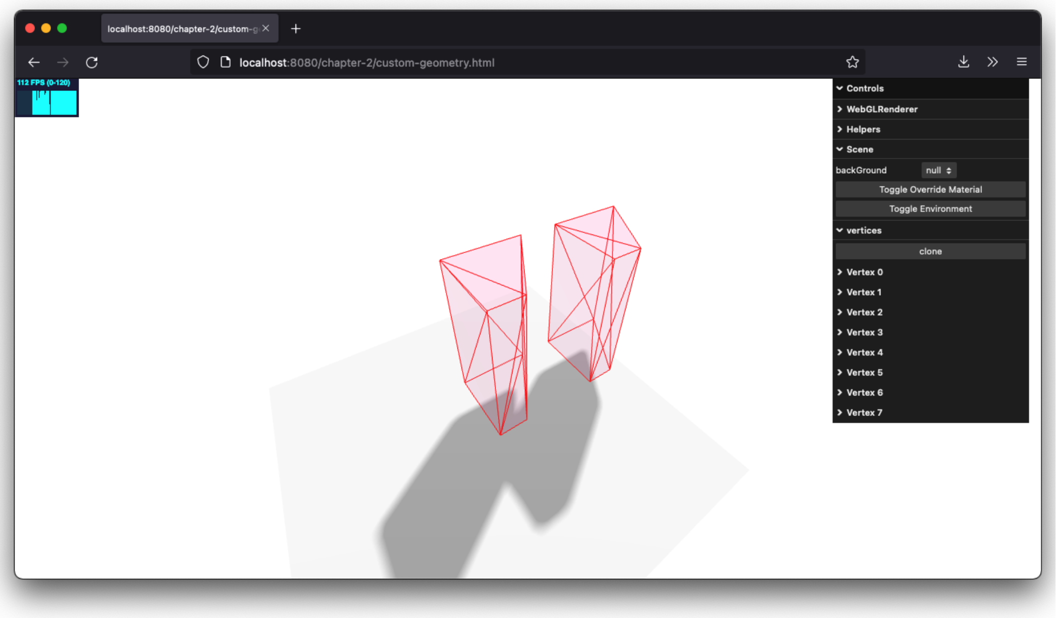
Figure 2.8 – Move vertices around to change the shape

This example, which uses the same setup as all our other examples, has a render loop. Whenever you change one of the properties in the drop-down control box, the cube is rendered based on the changed position of one of the vertices. This isn't something that works out of the box. For performance reasons, Three.js assumes that the geometry of a mesh won't change during its lifetime. For most geometries and use cases, this is a very valid assumption. If you, however, change the backing array (in this case, the `const faces = new Float32Array([...])` array), we need to tell Three.js that something has changed. You can do that by setting the `needsUpdate` property of the relevant attribute to `true`. This will look something like the following:

```
mesh.geometry.attributes.position.needsUpdate = true;
mesh.geometry.computeVertexNormals();
```

Note that in the case of updated vertices, it is also a good idea to recalculate the normal vectors, to make sure the materials are also rendered correctly. More information on what a normal vector is and why it is important will be explained in *Chapter 10*, *Loading and Working with Textures*.

There is one button from the `chapter-2/custom-geometry` menu that we haven't addressed yet. In the menu on the right, there is a **clone** button. We mentioned that the geometry defines the form and shape of an object, and combined with a material, we create an object that can be added to the scene to be rendered by Three.js. With the `clone()` function, as the name implies, we can make a copy of the geometry and, for instance, use it to create a different mesh with a different material. In the same example, `chapter-2/custom-geometry`, you can see a **clone** button at the top of the control GUI, as in the following screenshot:


localhost:8080/chapter-2/custom-g
localhost:8080/chapter-2/custom-geometry.html
112 FPS (0-120)
Controls
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
vertices
clone
Vertex 0
Vertex 1
Vertex 2
Vertex 3
Vertex 4
Vertex 5
Vertex 6
Vertex 7

backGround null

Figure 2.9 – Make a clone of the geometry

If you click on this button, a clone (copy) will be made of the geometry as it currently is; then, a new object will be created with a different material, and finally, the object will be added to the scene. The code for this is as follows:

```
const cloneGeometry = (scene) => {
  const clonedGeometry = bufferGeometry.clone();
  const backingArray = clonedGeometry.getAttribute
    ("position").array;
  // change the position of the x vertices so it is placed
  // next to the original object
  for (const i in backingArray) {
    if ((i + 1) % 3 === 0) {
      backingArray[i] = backingArray[i] + 3;
    }
  }
  clonedGeometry.getAttribute("position").needsUpdate =
    true;
  const cloned = meshFromGeometry(clonedGeometry);
  cloned.name = "clonedGeometry";

  const p = scene.getObjectByName("clonedGeometry");
  if (p) scene.remove(p);

  scene.add(cloned);
};
```

As you can see in the preceding code, we use the `clone()` function to clone `bufferGeometry`. Once cloned, we make sure to update the `x` value of each vertex so the clone is put in a different position than the original one (we could also have used `translateX`, which we explain in the following section of this chapter). Next, we create a `THREE.Mesh`, remove the cloned mesh if it is there, and add the new clone. To create the new mesh, we use a custom function called `meshFromGeometry`. As a quick sidestep, let's look at how that is implemented as well:

```
const meshFromGeometry = (geometry) => {
  var materials = [
    new THREE.MeshBasicMaterial({ color: 0xff0000,
      wireframe: true }),
    new THREE.MeshLambertMaterial({
      opacity: 0.1,
      color: 0xff0044,
      transparent: true,
    }),
  ];
  var mesh = createMultiMaterialObject(geometry, materials);
  mesh.name = "customGeometry";
  mesh.children.forEach(function (e) {
    e.castShadow = true;
  });
  return mesh;
};
```

If you look back at this example, you can see a transparent cube and the lines (the edges) that make up our geometry. To do this, we create a multi-material mesh. This means that we tell Three.js to use two different materials in a single mesh. For this, Three.js provides a nice helper function called `createMultiMaterialObject`, which does what the name implies. Based on a geometry and a list of materials, it creates an object that we can add to the scene. There is one thing you need to know though when working with the results from a `createMultiMaterialObject` call. What you get back isn't a single mesh; it is a `THREE.Group`, a container object that, in this case, contains a separate `THREE.Mesh` for each of the materials we provide. So, when rendering the mesh, it looks like a single object, but it actually comprises multiple `THREE.Mesh` objects rendered on top of one another. This also means that if we want to have shadows, we need to enable this for each of the meshes inside the group (which is what we did in the preceding code fragment).

In the preceding code, we used `createMultiMaterialObject` from the `THREE.SceneUtils` object to add a wireframe to the geometry we created. Three.js also provides an alternative way of adding a wireframe using `THREE.WireframeGeometry`. Assuming you have a geometry called `geom`, you can create a wireframe geometry from that: `const wireframe = new THREE.WireframeGeometry(geom);`. Next, you can draw the lines of this geometry, using the `Three.LineSegments` object, by first creating a `const line = new THREE.LineSegments(wireframe)` object, and then adding it to the scene: `scene.add(line)`. Since this helper internally is just a `THREE.Line` object, you can style how the wireframe appears. For instance, to set the width of the wireframe lines, use `line.material.linewidth = 2;`.

We've already looked a bit at the `THREE.Mesh` object. In the next section, we'll dive a bit deeper into what you can do with it.

Functions and attributes for meshes

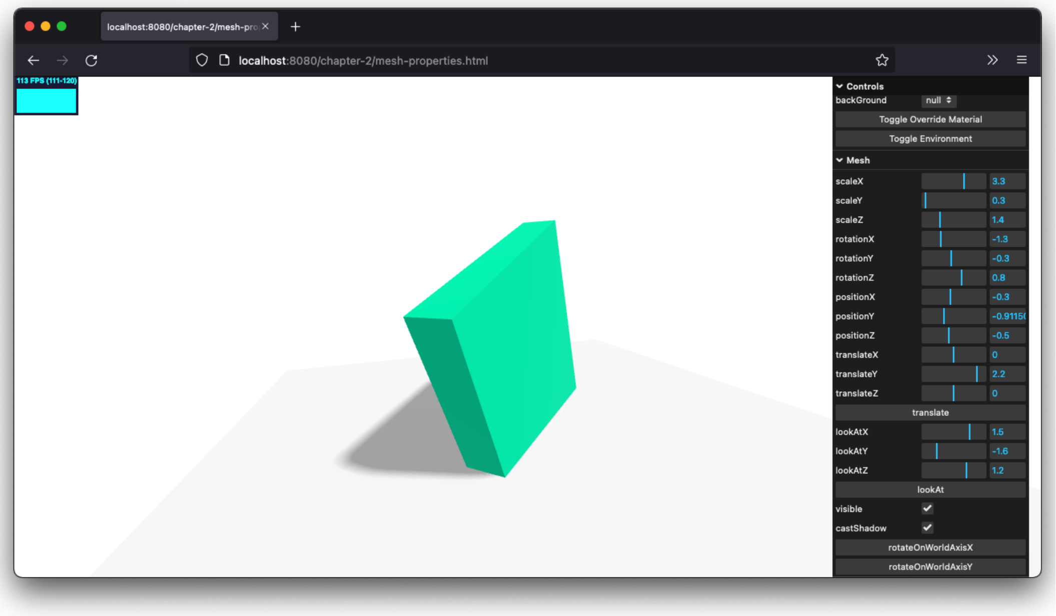
We've already learned that to create a mesh, we need a geometry and one or more materials. Once we have a mesh, we add it to the scene and it's rendered. There are a couple of properties you can use to change where and how this mesh appears on the scene. In our first example, we'll look at the following set of properties and functions:

- `position`: This determines the position of the object relative to the position of its parent. Most often, the parent of an object is a `THREE.Scene` object or a `THREE.Group` object.
- `rotation`: With this property, you can set the rotation of an object around any of its own axes. Three.js also provides specific functions for rotations around a single axis: `rotateX()`, `rotateY()`, and `rotateZ()`.
- `scale`: This property allows you to scale the object around its `x-`, `y-`, and `z-`axes.

- `translateX()` / `translateY()` and `translateZ()`: This property moves the object by a specified amount along the corresponding axis.
- `lookAt()`: This property points the object to a specific vector in space. This is an alternative to setting the rotation manually.
- `visible`: This property determines whether this mesh should be rendered or not.
- `castShadow`: This property determines whether this mesh casts shadows when it is hit by light. By default, meshes don't cast shadows.

When we're rotating an object, we're rotating around an axis. In a 3D scene, there are multiple spaces that have an axis you can rotate around. The `rotateN()` functions rotate the object around the axis in *local* space. This means the object rotates around the axis of its parent. So, when you add an object to the scene, the `rotateN()` functions will rotate that object around the main axis of the scene. When it is part of a nested group, these functions will rotate the object around the axis of its parent, which is normally the behavior you're looking for. Three.js also has a specific `rotateOnWorldAxis`, which allows you to rotate an object around the axis of the main `THREE.Scene` regardless of the actual parent of the object. Finally, you can also force the object to rotate around its own axis (this is called *object* space) by calling the `rotateOnAxis` function.

As always, we have an example ready for you that will allow you to play around with these properties. If you open `chapter-2/mesh-properties` in your browser, you get a drop-down menu where you can alter all these properties and directly see the result, as shown in the following screenshot:


localhost:8080/chapter-2/mesh-pro
localhost:8080/chapter-2/mesh-properties.html
113 FPS (111-120)
Controls
Toggle Override Material
Toggle Environment
Mesh
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY

backGround null
scaleX 3.3
scaleY 0.3
scaleZ 1.4
rotationX -1.3
rotationY -0.3
rotationZ 0.8
positionX -0.3
positionY -0.9115
positionZ -0.5
translateX 0
translateY 2.2
translateZ 0
lookAtX 1.5
lookAtY -1.6
lookAtZ 1.2
visible ☑
castShadow ☑

Figure 2.10 – Mesh properties

Let me walk you through the properties; I'll start with the `position` property.

Setting the location of the mesh with the position property

We've already seen this property a couple of times, so let's quickly address it. With this property, you set the `x-`, `y-`, and `z-`coordinates of the object relative to its parent. We'll get back to this in *Chapter 5*, *Learning to Work with Geometries*, when we look at grouping objects. We can set an object's position property in three different ways. We can set each coordinate directly:

```
cube.position.x = 10;
cube.position.y = 3;
cube.position.z = 1;
```

However, we can also set all of them at once, as follows:

```
cube.position.set(10,3,1);
```

There is also a third option. The `position` property is a `THREE.Vector3` object. That means we can also do the following to set this object:

```
cube.position = new THREE.Vector3(10,3,1)
```

Next on the list is the `rotation` property. You've already seen this property being used a couple of times here and in *Chapter 1*, *Creating Your First 3D Scene with Three.js*.

Defining the rotation of the mesh with the rotation property

With this property, you set the rotation of the object around one of its axes. You can set this value in the same manner as we did the position. A complete rotation, as you might remember from math class, is `2π`. You can configure this in Three.js in a couple of different ways:

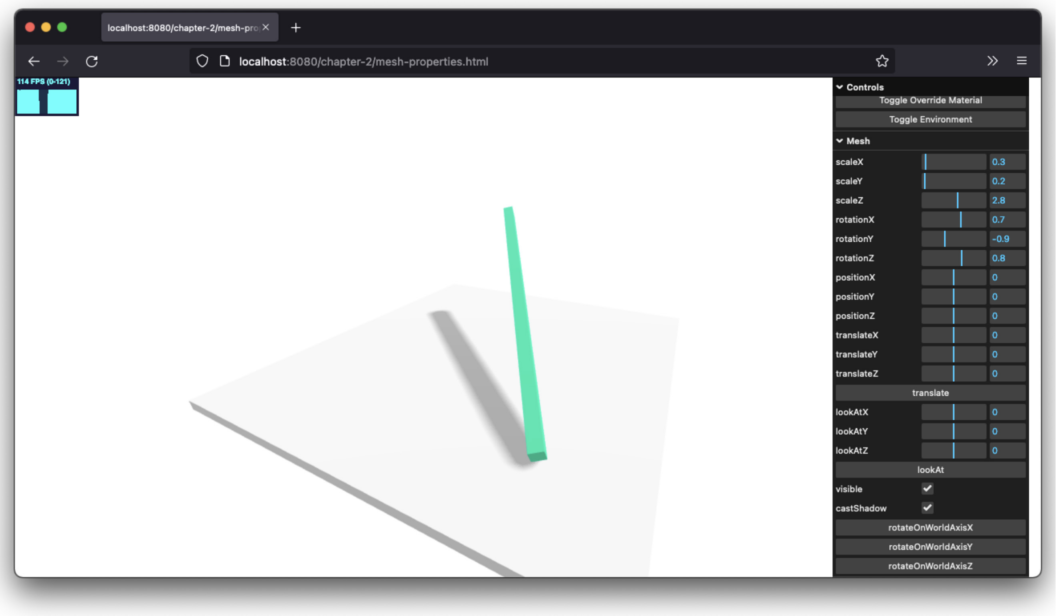
```
cube.rotation.x = 0.5*Math.PI;
cube.rotation.set(0.5*Math.PI, 0, 0);
cube.rotation = new THREE.Vector3(0.5*Math.PI,0,0);
```

If you want to use degrees (from 0 to 360) instead, we'll have to convert those to radians. This can be easily done as follows:

```
const degrees = 45;
const inRadians = degrees * (Math.PI / 180);
```

In the preceding code block, we've done the conversion ourselves. Three.js also provides the `MathUtils` class, which provides a lot of helpful conversions, including one that does the same thing as we did in the preceding code block. You can play around with this property using the `chapter-2/mesh-properties` example.

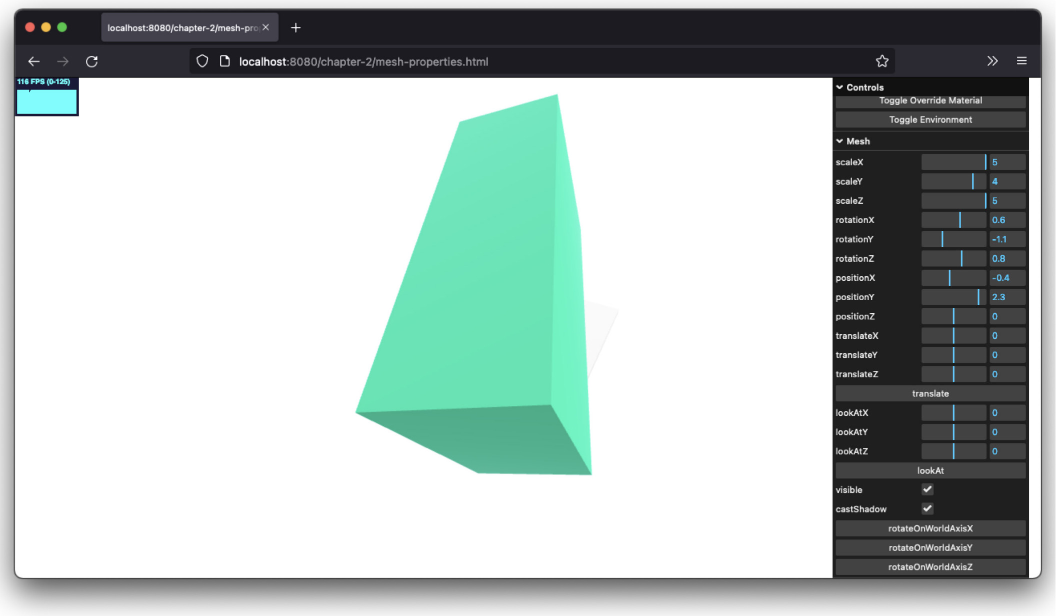
The next property on our list is one we haven't talked about: `scale`. The name pretty much sums up what you can do with this property. You can scale the object along a specific axis. If you set the scale to a value less than one, the object will shrink, as shown in the following screenshot:


localhost:8080/chapter-2/mesh-pro
localhost:8080/chapter-2/mesh-properties.html
114 FPS (0-121)
Controls
Toggle Override Material
Toggle Environment
Mesh
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ

scaleX 0.3
scaleY 0.2
scaleZ 2.8
rotationX 0.7
rotationY -0.9
rotationZ 0.8
positionX 0
positionY 0
positionZ 0
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑
castShadow ☑

Figure 2.11 – Use scale to shrink a mesh

When you use values greater than one, the object will become larger, as shown in the following screenshot:


localhost:8080/chapter-2/mesh-pro
localhost:8080/chapter-2/mesh-properties.html
116 FPS (0-125)
Controls
Toggle Override Material
Toggle Environment
Mesh
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ

scaleX	5
scaleY	4
scaleZ	5
rotationX	0.6
rotationY	-1.1
rotationZ	0.8
positionX	-0.4
positionY	2.3
positionZ	0
translateX	0
translateY	0
translateZ	0
lookAtX	0
lookAtY	0
lookAtZ	0
visible	☑
castShadow	☑

Figure 2.12 – Use scale to grow a mesh

The next part of the mesh that we'll look at is the `translate` property.

Changing the position using the translate property

With `translate`, you can also change the position of an object, but instead of defining the absolute position where you want the object to be, you define the distance the object should move, relative to its current position. For instance, we have a sphere that is added to a scene, and its position has been set to `(1, 2, 3)`. Next, we translate the object along its `x`-axis: `translateX(4)`. Its position will now be `(5, 2, 3)`. If we want to restore the object to its original position, we use `translateX(-4)`. In the `chapter-2/mesh-properties` example, there is a menu tab called **translate**. From there, you can experiment with this functionality. Just set the `translate` values for `x`, `y`, and `z` and hit the **translate** button. You'll see the object being moved to a new position based on these three values.

The last two properties we'll look at are used to remove the object completely, by setting the `visible` property to `false`, and disabling whether this object casts shadows by setting the `castShadow` property to `false`. When you click on these buttons, you'll see the cube becomes invisible and visible, and you can disable it from casting shadows.

For more information on meshes, geometries, and what you can do with these objects, check out *Chapter 5*, *Learning to Work with Geometries*, and *Chapter 7*, *Points and Sprites*.

So far, we've looked at `THREE.Scene`, the main object that holds all the objects that we want to render, and we've looked in detail at what a `THREE.Mesh` is, and how you can create a `THREE.Mesh` and position it in a scene. In previous sections, we've already used a camera to determine what part of `THREE.Scene` you wanted to render but haven't explained in detail yet how you can configure a camera. In the next section, we'll dive into those details.

Using different cameras for different scenes

There are two different camera types in Three.js: the orthographic camera and the perspective camera. Note that Three.js also provides a couple of very specific cameras for creating scenes that can be viewed using 3D glasses or VR gear. We won't go into detail about those cameras in this book, since they work exactly the same as the cameras explained in this chapter. If you're interested in these cameras, Three.js provides a few standard examples:

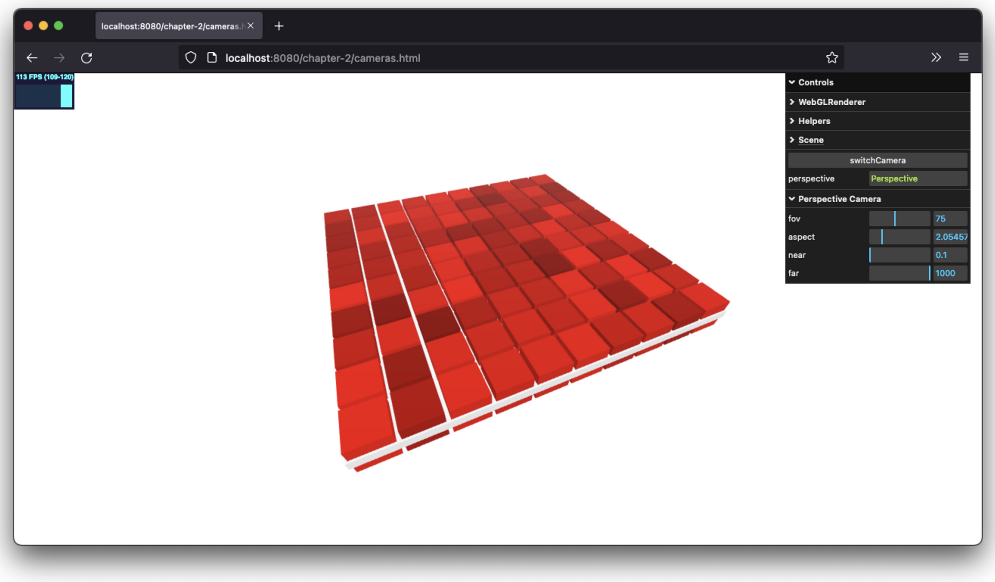
- **Anaglyph effect**: `https://threejs.org/examples/#webgl_effects_anaglyph`
- **Parallax barrier**: `https://threejs.org/examples/#webgl_effects_parallaxbarrier`
- **Stereo effect**: `https://threejs.org/examples/#webgl_effects_stereo`

If you're looking for simple VR cameras, you can use `THREE.StereoCamera` to create 3D scenes that are rendered side to side (standard stereo effect), use a parallel barrier (as 3DS provides), or provide an anaglyph effect where the different views are rendered in different colors. Alternatively, Three.js has some experimental support for the WebVR standard, which is supported by a number of browsers (for more info, see `https://webvr.info/developers/`). To use this, not that much needs to change. You just set `renderer.vr.enabled = true`, and Three.js will handle the rest. The Three.js website has a couple of examples where this property and some other features of Three.js's support for WebVR are demonstrated: `https://threejs.org/examples/`.

For now, we'll focus on the standard perspective and orthographic cameras. The best way to explain the differences between these cameras is by looking at a couple of examples.

An orthographic camera versus a perspective camera

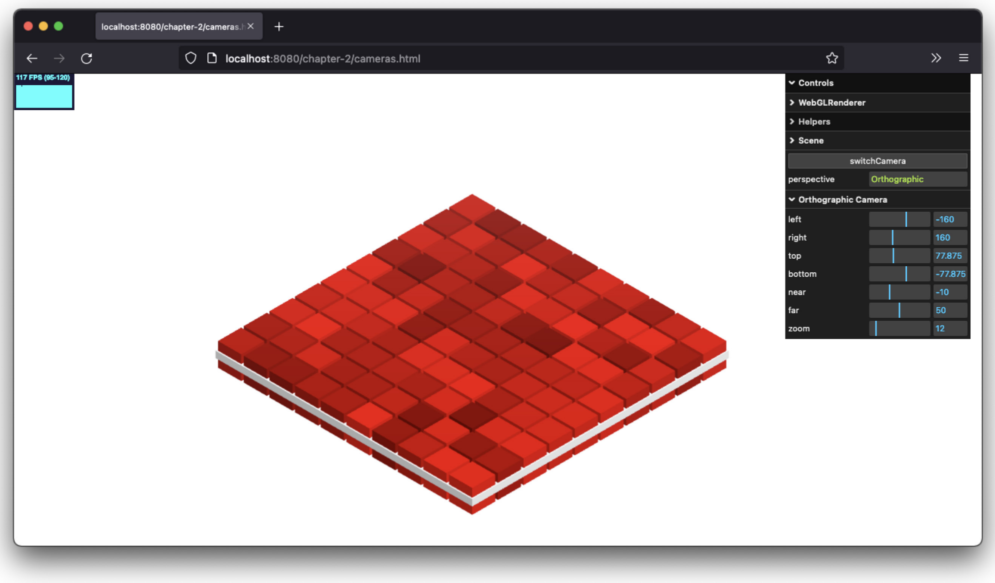
In the examples for this chapter, you can find a demo called `chapter2/cameras`. When you open this example, you'll see something like the following:


localhost:8080/chapter-2/cameras.h
localhost:8080/chapter-2/cameras.html
113 FPS (109-120)
Controls
WebGLRenderer
Helpers
Scene
switchCamera
Perspective Camera

perspective Perspective
fov 75
aspect 2.05457
near 0.1
far 1000

Figure 2.13 – Perspective camera view

This preceding screenshot is called a perspective view and is the most natural view. As you can see from this figure, the farther away the cubes are from the camera, the smaller they are rendered. If we change the camera to the other type supported by Three.js, the orthographic camera, you'll see the following view of the same scene:


localhost:8080/chapter-2/cameras.h
localhost:8080/chapter-2/cameras.html
117 FPS (95-120)
Controls
WebGLRenderer
Helpers
Scene
switchCamera
Orthographic Camera

perspective Orthographic
left -160
right 160
top 77.875
bottom -77.875
near -10
far 50
zoom 12

Figure 2.14 – Orthographic camera view

With the orthographic camera, all the cubes are rendered the same size; the distance between an object and the camera doesn't matter. This is often used in 2D games, such as old versions of *Civilization* and *SimCity 4*:

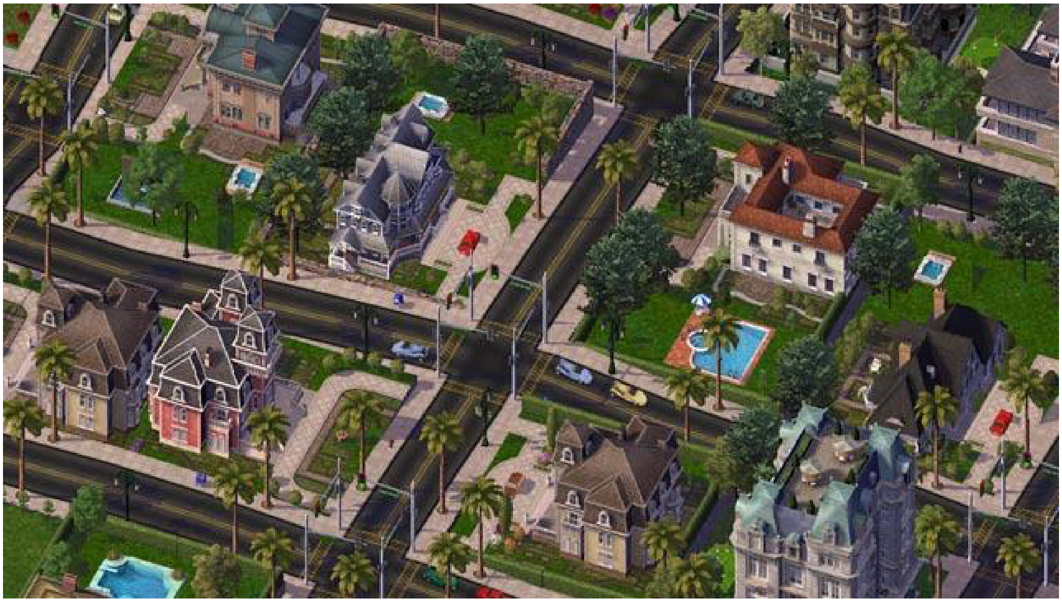


Figure 2.15 – Orthographic usage in SimCity 4

Perspective camera properties

Let's first look a bit closer at `THREE.PerspectiveCamera`. In the example, you can set a number of properties that define what is shown through the lens of the camera:

- `fov`: The **Field of View** (**FOV**) is the part of the scene that can be seen from the position of the camera. Humans, for instance, have an almost 180-degree FOV, while some birds even have a complete 360-degree FOV. But since a normal computer screen doesn't completely fill our vision, a smaller value is often chosen. Generally, for games, a FOV of between 60 and 90 degrees is chosen. *Good default*: 50
- `aspect`: This is the aspect ratio between the horizontal and vertical sizes of the area where we're rendering the output. In our case, since we use the entire window, we just use that ratio. The aspect ratio determines the difference between the horizontal FOV and the vertical FOV. *Good default*: `window.innerWidth / window.innerHeight`
- `near`: The `near` property defines how close to the camera Three.js should render the scene. Normally, we set this to a very small value to directly render everything from the position of the camera. *Good default*: 0.1

- `far`: The `far` property defines how far the camera can see from the position of the camera. If we set this too low, part of our scene might not be rendered, and if we set it too high, in some cases, it might affect the rendering performance. *Good default*: 100
- `zoom`: The `zoom` property allows you to zoom in and out of the scene. When you use a number lower than 1, you zoom out of the scene, and if you use a number higher than 1, you zoom in. Note that if you specify a negative value, the scene will be rendered upside down. *Good default*: 1

The following diagram gives a good overview of how these properties work together to determine what you see:

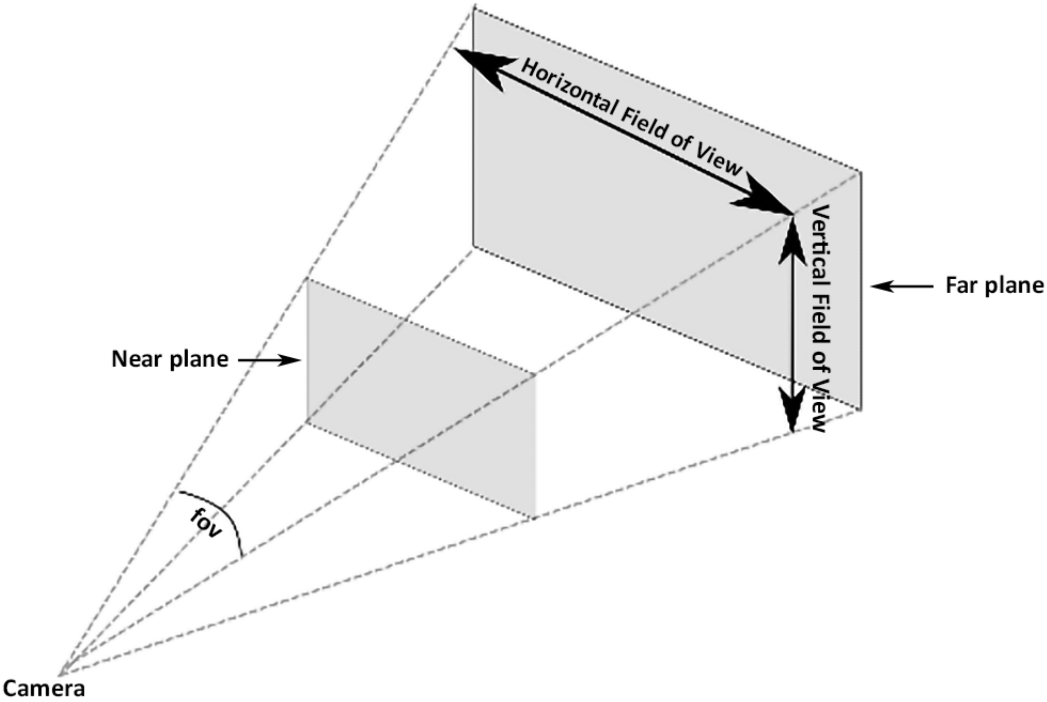
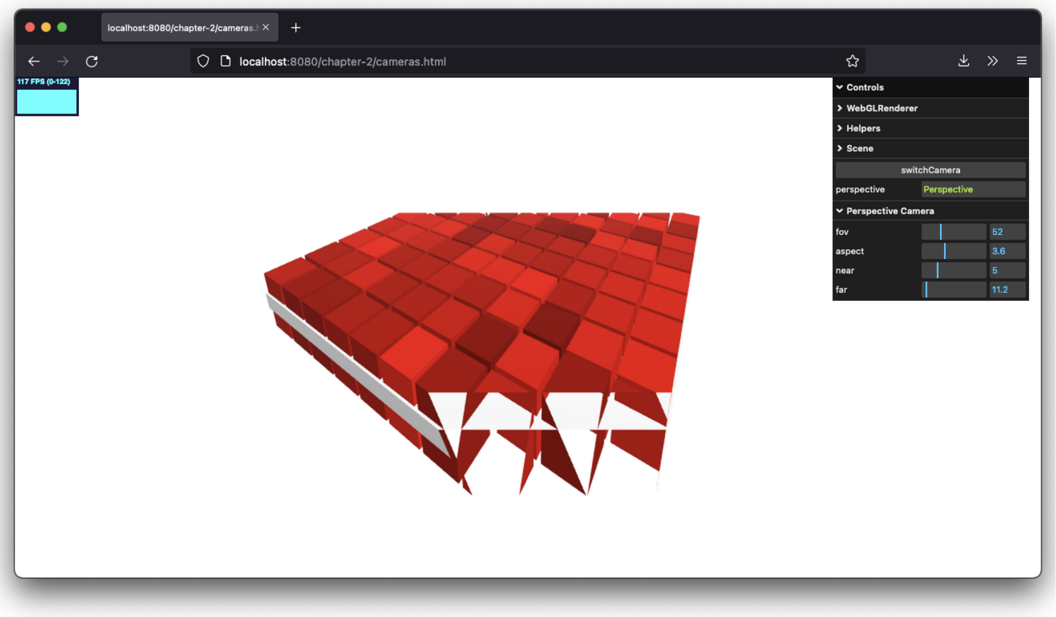


Figure 2.16 – Properties of the perspective camera

The `fov` property of the camera determines the horizontal FOV. Based on the `aspect` property, the vertical FOV is determined. The `near` property is used to determine the position of the near plane, and the `far` property determines the position of the far plane. The area between the near plane and the far plane will be rendered as follows:



perspective Perspective

fov 52

aspect 3.6

near 5

far 11.2

Figure 2.17 – Far and near clip the rendered mesh

Orthographic camera properties

To configure the orthographic camera, we need to use other properties. The orthographic projection isn't interested in which aspect ratio to use or what FOV we have of the scene, since all the objects are rendered at the same size. When you define an orthographic camera, you define the cuboid area that needs to be rendered. The properties of the orthographic camera reflect this, as follows:

- `left`: This is described in the Three.js documentation as the camera frustum left plane. You should see this as the left-hand border of what will be rendered. If you set this value to `-100`, you won't see any objects that are positioned farther than that on the left-hand side.
- `right`: The `right` property works in a way similar to the `left` property, but this time, on the other side of the screen. Anything farther to the right won't be rendered.
- `top`: This is the top position to be rendered.
- `bottom`: This is the bottom position to be rendered.
- `near`: From this point, based on the position of the camera, the scene will be rendered.
- `far`: To this point, based on the position of the camera, the scene will be rendered.

- `zoom`: This allows you to zoom in and out of the scene. When you use a number lower than `1`, you'll zoom out of the scene; if you use a number higher than `1`, you'll zoom in. Note that if you specify a negative value, the scene will be rendered upside down. The default value is `1`.

And all these properties can be summarized in the following diagram:

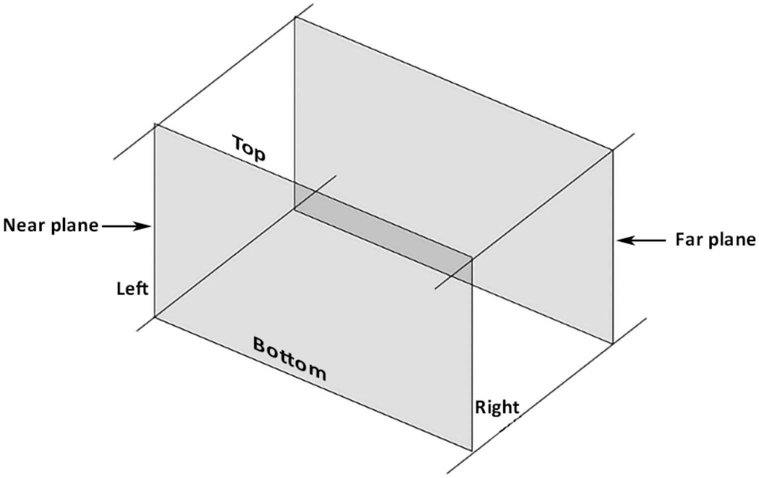
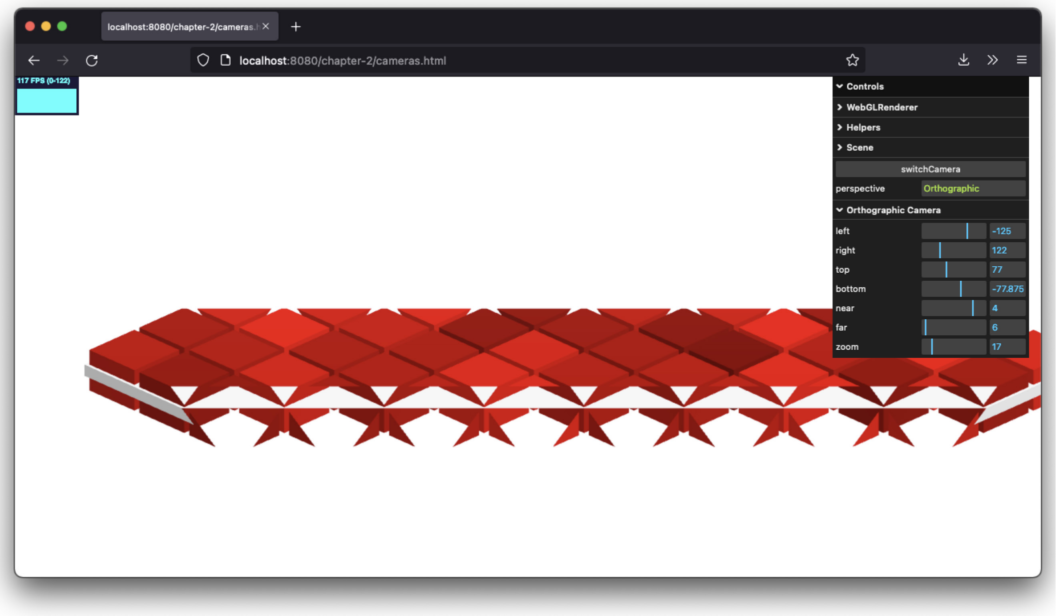

Top
Near plane
Far plane
Left
Bottom
Right

Figure 2.18 – Properties of the orthographic camera

And just like with the perspective camera, you can exactly define the area of the scene you want to render:


localhost:8080/chapter-2/cameras.h
localhost:8080/chapter-2/cameras.html
117 FPS (0-122)
Controls
WebGLRenderer
Helpers
Scene
switchCamera

perspective Orthographic

Orthographic Camera

left -125
right 122
top 77
bottom -77.875
near 4
far 6
zoom 17

Figure 2.19 – Clipped area with an orthographic camera

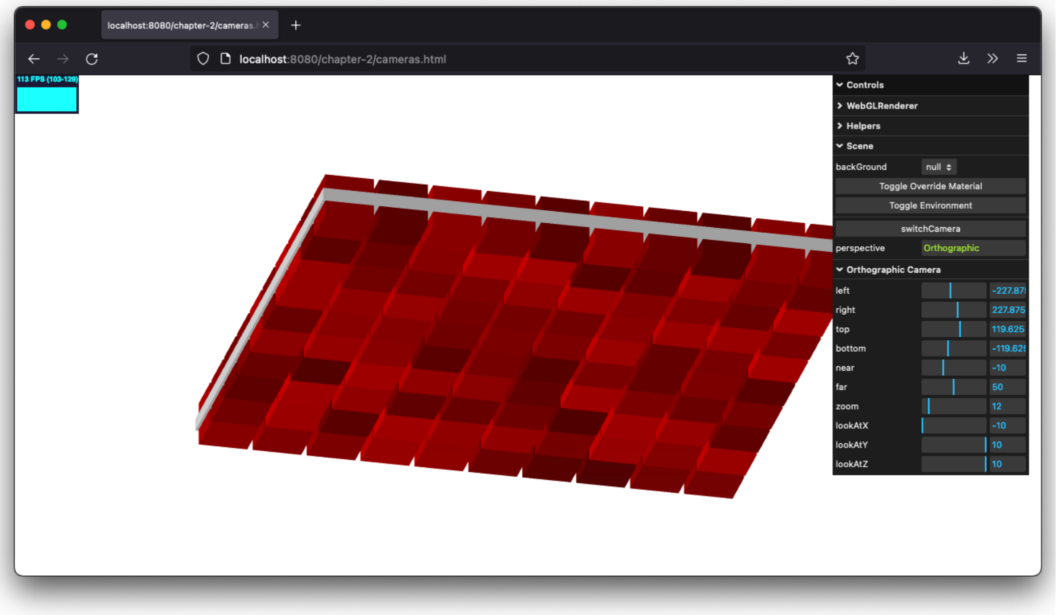
In the previous section, we explained the different cameras supported by Three.js. You've learned how to configure them, and how you can use their properties to render different parts of the scene. What we didn't show yet is how you can control what part of the scene a camera is looking at. We'll explain that in the next section.

Looking at specific points

So far, you've seen how to create a camera and what the various arguments mean. In *Chapter 1*, *Creating Your First 3D Scene with Three.js*, you also saw that you need to position your camera somewhere in the scene and that the view from that camera is rendered. Normally, the camera is pointed to the center of the scene: position `(0, 0, 0)`. We can, however, easily change what the camera is looking at, as follows:

```
camera.lookAt(new THREE.Vector3(x, y, z));
```

In the `chapter2/cameras` example, you can also specify the coordinates you want the camera to look at. Note that when you change `lookAt` in the `OrthographicCamera` settings, the cubes still stay the same size.


localhost:8080/chapter-2/cameras.
localhost:8080/chapter-2/cameras.html
113 FPS (103-129)
Controls
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
switchCamera
Orthographic Camera

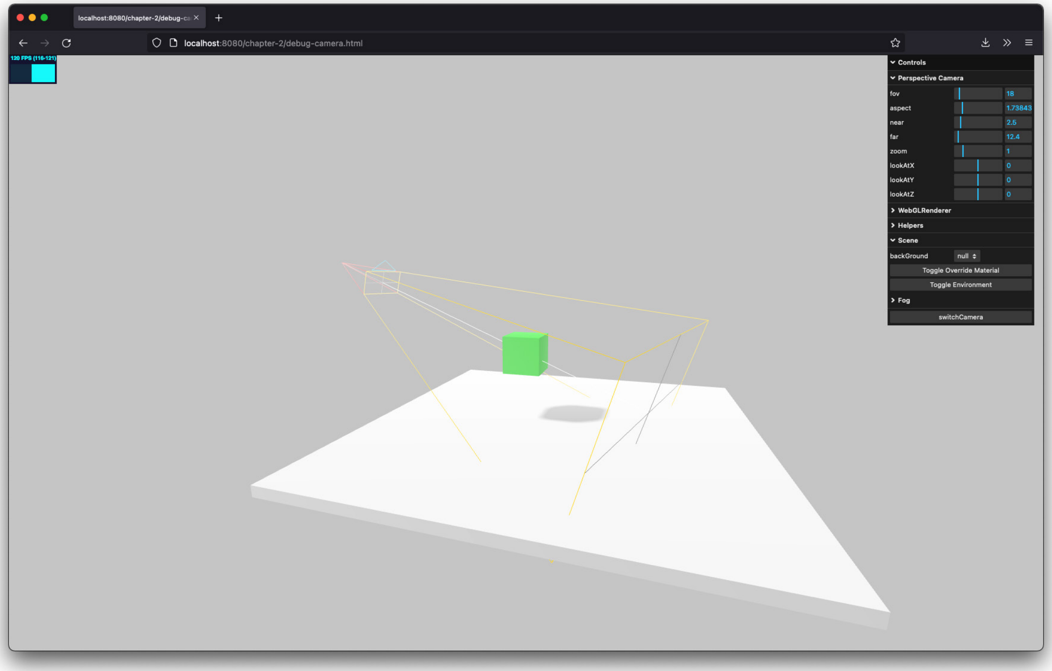
backGround null
perspective Orthographic
left -227.87
right 227.875
top 119.625
bottom -119.62
near -10
far 50
zoom 12
lookAtX -10
lookAtY 10
lookAtZ 10

Figure 2.20 – Changed lookAt property for the orthographic camera

When you use the `lookAt` function, you point the camera at a specific position. You can also use this to make the camera follow an object around a scene. Since every `THREE.Mesh` object has a position that is a `THREE.Vector3` object, you can use the `lookAt` function to point to a specific mesh in the scene. All you need to use is this: `camera.lookAt(mesh.position)`. If you call this in the render loop, you'll make the camera follow an object as it moves through a scene.

Debugging what a camera looks at

When looking at configuring the camera, having a menu where you can play around with the different settings can help a lot. Sometimes, though, you might want to exactly see the area that will be rendered by the camera. Three.js allows you to do this, by visualizing the frustum of the camera (the area that is shown by the camera). To do this, we simply add an additional camera to the scene and add a camera helper. To see this in action, open the `chapter-2/debug-camera.html` example:


localhost:8080/chapter-2/debug-ca
localhost:8080/chapter-2/debug-camera.html
120 FPS (116-121)
Controls
Perspective Camera
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
Fog
switchCamera

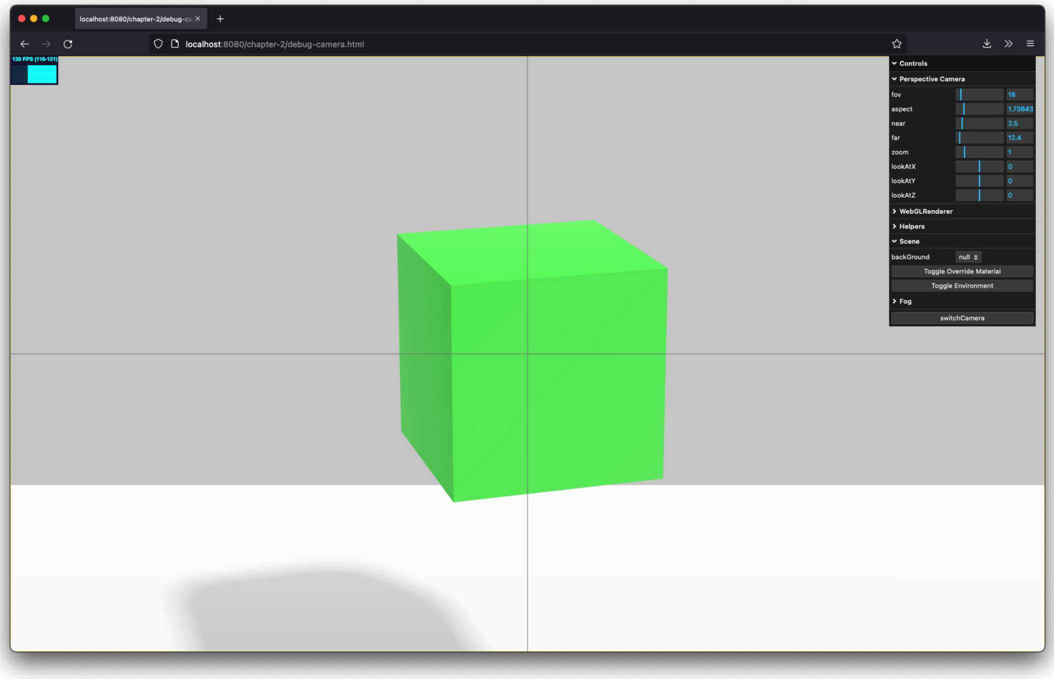
fov 18
aspect 1.73843
near 2.5
far 12.4
zoom 1
lookAtX 0
lookAtY 0
lookAtZ 0
backGround null

Figure 2.21 – Show the frustum of the camera

In the preceding figure, you can see the outline of the frustum of a perspective camera. If you change the properties in the menu, you can see that the frustum changes as well. This frustum is visualized by adding the following:

```
const helper = new THREE.CameraHelper(camera);
scene.add(helper);
// in the render loop
helper.update();
```

We've also added a **switchCamera** button, which allows you to switch between the external camera looking in at the scene and the main camera in the scene. This provides a great way to get the correct settings for your camera:


localhost:8080/chapter-2/debug-ca
localhost:8080/chapter-2/debug-camera.html
120 FPS (116-121)
Controls
Perspective Camera
WebGLRenderer
Helpers
Scene
Toggle Override Material
Toggle Environment
Fog
switchCamera

fov 18
aspect 1.73843
near 2.5
far 12.4
zoom 1
lookAtX 0
lookAtY 0
lookAtZ 0
backGround null

Figure 2.22 – Switch between cameras

Switching cameras is really easy in Three.js. The only thing you need to do is tell Three.js you want to render the scene through a different camera.

Summary

We discussed a lot in this second introductory chapter. We showed the functions and properties of `THREE.Scene` and explained how you can use these properties to configure your main scene. We also showed you how to create geometries. You can either create them from scratch using a `THREE.Buffergeometry` object or use any of the built-in geometries Three.js provides. Finally, we showed you how to configure the two main cameras Three.js provides. `THREE.PerspectiveCamera` renders a scene using a real-world perspective, and `THREE.OrthographicCamera` provides the fake 3D effect often seen in games. We've also covered how geometries work in Three.js and you can now easily create your own geometries from the standard geometries provided by Three.js or by crafting them by hand.

In the next chapter, we'll look at the various light sources that are available in Three.js. You'll learn how the various light sources behave, how to create and configure them, and how they affect different materials.

3

Working with Light Sources in Three.js

In *Chapter 1*, *Creating Your First 3D Scene with Three.js*, you learned about the basics of Three.js, and in *Chapter 2*, *The Basic Components that Make up a Three.js Application*, we looked a bit deeper at the most important parts of the scene: the geometries, meshes, and cameras. You might have noticed that we skipped exploring the details of lights in that chapter, even though they make up an important part of every Three.js scene. Without lights, we won't see anything rendered (unless we use basic or wireframe materials). Since Three.js contains several different light sources, each of which has a specific use, we'll use this chapter to explain the various details of lights and prepare you for the upcoming chapters on material usage. By the end of this chapter, you'll know the differences between the available lights and be able to choose and configure the correct light for your scene.

Note

WebGL itself doesn't have inherent support for lighting. Without Three.js, you would have to write specific WebGL shader programs to simulate these kinds of lights, which is quite difficult. A good introduction to simulating lighting in WebGL from scratch can be found at `https://developer.mozilla.org/en-US/docs/Web/WebGL/Lighting_in_WebGL`.

In this chapter, we will cover the following topics:

- Different kinds of lighting in Three.js
- Working with basic lights
- Working with special lights

As with all the chapters, we have a lot of examples that you can use to experiment with the behavior of lights. The examples shown in this chapter can be found in the `chapter-03` folder of the supplied sources.

What lighting types are provided in Three.js?

Several different lights are available in Three.js that all have specific behavior and usages. In this chapter, we'll discuss the following set of lights:

- `THREE.AmbientLight`: This is a basic light, the color of which is added to the current color of the objects in the scene.
- `THREE.PointLight`: This is a single point in space from which light spreads in all directions. This light can be used to create shadows.
- `THREE.SpotLight`: This light source has a cone-like effect like that of a desk lamp, a spotlight in the ceiling, or a torch. This light can cast shadows.
- `THREE.DirectionalLight`: This is also called infinite light. The light rays from this light can be seen as parallel, similar to the light from the Sun. This light can also be used to create shadows.
- `THREE.HemisphereLight`: This is a special light and can be used to create more natural-looking outdoor lighting by simulating a reflective surface and a faintly illuminating sky. This light also doesn't provide any shadow-related functionality.
- `THREE.RectAreaLight`: With this light source, instead of a single point in space, you can specify an area from which light emanates. `THREE.RectAreaLight` doesn't cast any shadows.
- `THREE.LightProbe`: This is a special kind of light source where, based on the environment map used, a dynamic ambient light source is created to light the scene.
- `THREE.LensFlare`: This is not a light source, but with `THREE.LensFlare`, you can add a lens flare effect to the lights in your scene.

This chapter is divided into two main parts. First, we'll look at the basic lights: `THREE.AmbientLight`, `THREE.PointLight`, `THREE.SpotLight`, and `THREE.DirectionalLight`. All these lights extend the base `THREE.Light` object, which provides shared functionality. The lights mentioned here are simple lights that require little setup and can be used to recreate most of the required lighting scenarios. In the second part, we will look at a couple of special-purpose lights and effects: `THREE.HemisphereLight`, `THREE.RectAreaLight`, `THREE.LightProbe`, and `THREE.LensFlare`. You'll probably only need these lights in very specific cases.

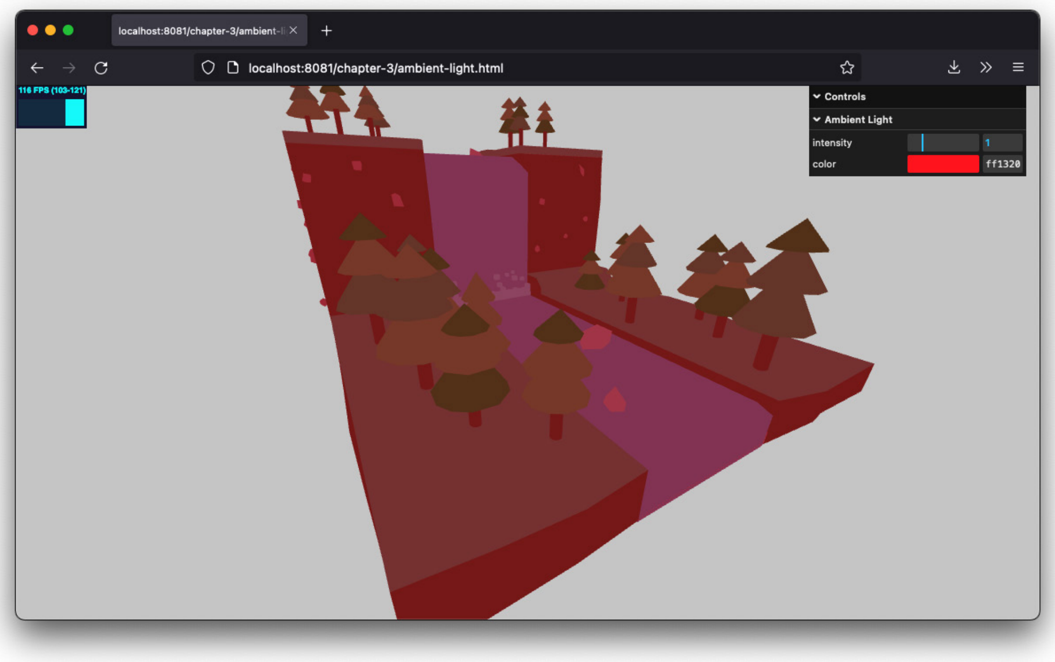
Working with basic lights

We'll start with the most basic of the lights: `THREE.AmbientLight`.

THREE.AmbientLight

When you create a `THREE.AmbientLight`, the color is applied globally. There isn't a specific direction this light comes from, and `THREE.AmbientLight` doesn't contribute to any shadows. You would normally not use `THREE.AmbientLight` as the single source of light in a scene since it applies its color to all the objects in the scene in the same way, regardless of the shape of the mesh. You use it together with other lighting sources, such as `THREE.SpotLight` or `THREE.DirectionalLight`, to soften the shadows or add some additional color to the scene. The easiest way to understand this is by looking at the `ambient-light.html` example in the `chapter-03` folder. In this example, you get a simple user interface that can be used to modify the `THREE.AmbientLight` object that is available in this scene.

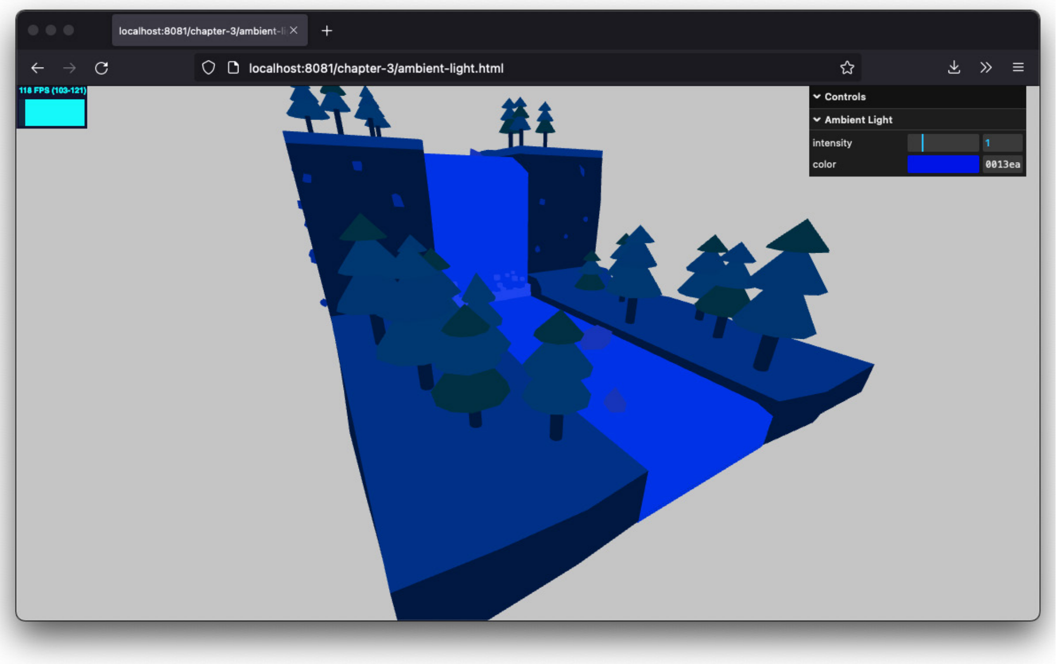
In the following screenshots, you can see that we used a simple waterfall model, and made the `color` and `intensity` properties of the used `THREE.AmbientLight` object configurable. In this first screenshot, you can see what happens when we set the color of the light to red:


localhost:8081/chapter-3/ambient-l
localhost:8081/chapter-3/ambient-light.html
116 FPS (103-121)
Controls
Ambient Light

intensity 1
color ff1320

Figure 3.1 – Ambient light set to red

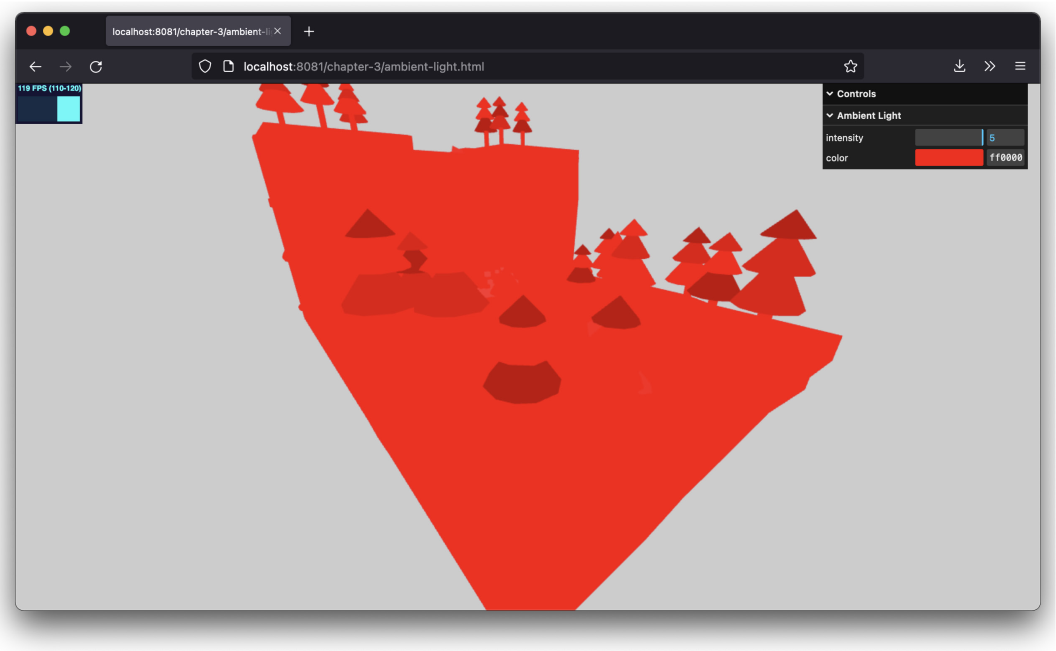
As you can see, each element in our scene now has a red color added to its original color. And if we change the color to blue, we'll get something like this:


localhost:8081/chapter-3/ambient-li
localhost:8081/chapter-3/ambient-light.html
118 FPS (103-121)
Controls
Ambient Light

intensity 1
color 0013ea

Figure 3.2 – Ambient light set to blue

As this screenshot shows, the blue color is applied to all the objects and casts a glow over the complete scene. What you should remember when working with this light is that you should be very conservative with the color you specify. If the color you specify is too bright, you'll quickly get a completely oversaturated image. Besides the color, we can also set the `intensity` property of the light. This property determines how much `THREE.AmbientLight` affects the colors in the scene. If we turn it down, only a little of the color is applied to the objects in the scene. If we turn it up, our scene becomes really bright:


localhost:8081/chapter-3/ambient-li
localhost:8081/chapter-3/ambient-light.html
119 FPS (110-120)
Controls
Ambient Light

intensity 5
color ff0000

Figure 3.3 – Ambient light set to red with high intensity

Now that we've seen what it does, let's look at how you can create and use a `THREE.AmbientLight`. The following lines of code show you how to create a `THREE.AmbientLight`:

```
const color = new THREE.Color(0xffffff);
const light = new THREE.AmbientLight(color);
scene.add(light);
```

Creating a `THREE.AmbientLight` is very simple and only takes a couple of steps. `THREE.AmbientLight` doesn't have a position and is applied globally, so we only need to specify the color and add this light to the scene. Optionally, we can also provide an additional value in this constructor for the intensity of this light. Since we didn't specify it here, it uses a default intensity of `1`.

Note that in the previous code fragment, we passed in an explicit `THREE.Color` object to the constructor of `THREE.AmbientLight`. We could have also passed in the color as a string – for example, `"rgb(255, 0, 0)"` or `"hsl(0, 100%, 50%)"` – or as a number, as we did in the previous chapters: `0xff0000`. More information on this can be found in the *Using the THREE.Color object* section.

Before we discuss `THREE.PointLight`, `THREE.SpotLight`, and `THREE.DirectionalLight`, first, let's highlight their main difference – that is, how they emit light. The following diagram shows how these three light sources emit light:

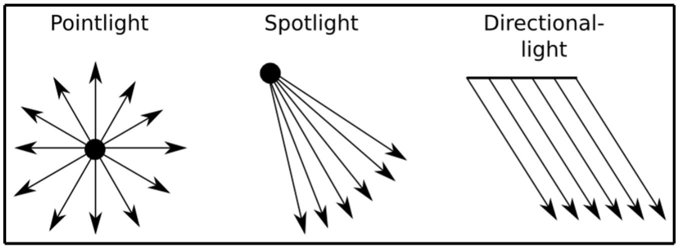


Figure 3.4 – How different light sources emit light

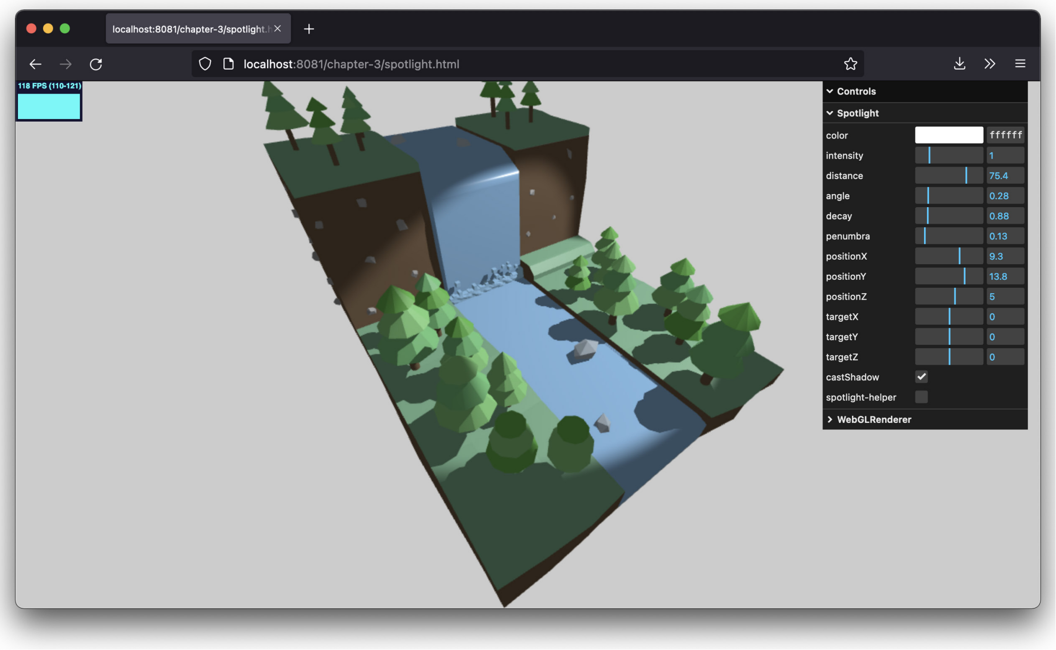
You can see the following from this diagram:

- `THREE.PointLight` emits light from a specific point in all directions
- `THREE.SpotLight` emits light from a specific point in a cone-like shape
- `THREE.DirectionalLight` doesn't emit light from a single point but emits light rays from a 2D plane, where the rays are parallel to each other

We'll look at these light sources in more detail in the next few sections. Let's start with `THREE.SpotLight`.

THREE.SpotLight

`THREE.SpotLight` is one of the lights you'll use often (especially if you want to use shadows). `THREE.SpotLight` is a light source that has a cone-like effect. You can compare this with a flashlight or a lantern. This light source has a direction and an angle at which it produces light. The following screenshot shows what a `THREE.SpotLight` looks like (`spotlight.html`):


localhost:8081/chapter-3/spotlight.h
localhost:8081/chapter-3/spotlight.html
118 FPS (110-121)
Controls
Spotlight
WebGLRenderer

color ffffff
intensity 1
distance 75.4
angle 0.28
decay 0.88
penumbra 0.13
positionX 9.3
positionY 13.8
positionZ 5
targetX 0
targetY 0
targetZ 0
castShadow ☑
spotlight-helper ☐

Figure 3.5 – Spotlight lighting a scene

The following table lists all the properties that you can use to finetune `THREE.SpotLight`. First, we'll look at the properties specific to the light's behavior:

Name	**Description**
`Angle`	Determines how wide the beam emerging from the light is. The width is measured in radians and defaults to `Math.PI/3`.
`castShadow`	If set to `true`, the light to which the property is applied will create shadows. See the following table on how to configure the shadows.
`Color`	Indicates the color of the light.
`decay`	Indicates the amount the light intensity diminishes the farther you move away from the light source. A `decay` of `2` leads to more realistic light, and the default value is `1`. This property is only effective when the `physicallyCorrectLights` property is set on `WebGLRenderer`.
`distance`	When this property is set to a non-`0` value, the light intensity will decrease linearly from the set intensity at the light's position to `0` at the specified distance.
`intensity`	Indicates the intensity with which the light shines. The property's default value is `1`.
`penumbra`	Indicates the percentage at the edge of the spotlight's coin, which is smoothed (blurred) to `0`. It takes a value between `0` and `1`, where the default is `0`.
`power`	Denotes the light's `power` when rendered in the physically correct mode (enable this by setting the `physicallyCorrectLights` property set on `WebGLRenderer`). This property is measured in lumens and the default value is `4*Math.PI`.
`position`	Indicates the position of a light in `THREE.Scene`.
`target`	With `THREE.SpotLight`, the direction the light is pointed in is important. With the `target` property, you can point `THREE.SpotLight` to look at a specific object or position in the scene. Note that this property requires a `THREE.Object3D` object (such as a `THREE.Mesh`). This is in contrast to the cameras we saw in *Chapter 2*, which use `THREE.Vector3` in their `lookAt` function.
`visible`	If this property is set to `true` (the default), the light is turned on, while if it is set to `false`, the light is turned off.

Figure 3.6 – Properties of the THREE.SpotLight object

When you enable the shadow for `THREE.SpotLight`, you can control how that shadow is rendered. You can control this through the shadow property of a `THREE.SpotLight`, which can comprise the following:

Name	**Description**
`shadow.bias`	Moves the cast shadow away or toward the object casting the shadow. You can use this to solve some strange effects when you work with very thin objects. If you see strange shadow effects on your models, small values (for example, `0.01`) for this property can often resolve the issue. The default value for this property is `0`.
`shadow.camera.far`	Determines from what distance from the light shadows should be created. The default value is `5000`. Note that you can also set all the other properties provided for `THREE.PerspectiveCamera`, which we showed in *Chapter 2*.
`shadow.camera.fov`	Determines how large the field of view used to create shadows is (see the *Using different cameras for different scenes* section in *Chapter 2*). The default value is `50`.
`shadow.camera.near`	Determines from what distance from the light shadows should be created. The default value is `50`.
`shadow.mapSize.width` and `shadow.mapSize.height`	Determine how many pixels are used to create a shadow. Increase these when a shadow has jagged edges or doesn't look smooth. This can't be changed after the scene has been rendered. The default value for both is `512`.
`shadow.radius`	When this value is set higher than 1, the edge of the shadows will be blurred. This won't have any effect if the `shadowMap.type` property of `THREE.WebGlRenderer` is set to `THREE.BasicShadowMap`.

Figure 3.7 – Shadow properties of the THREE.SpotLight object

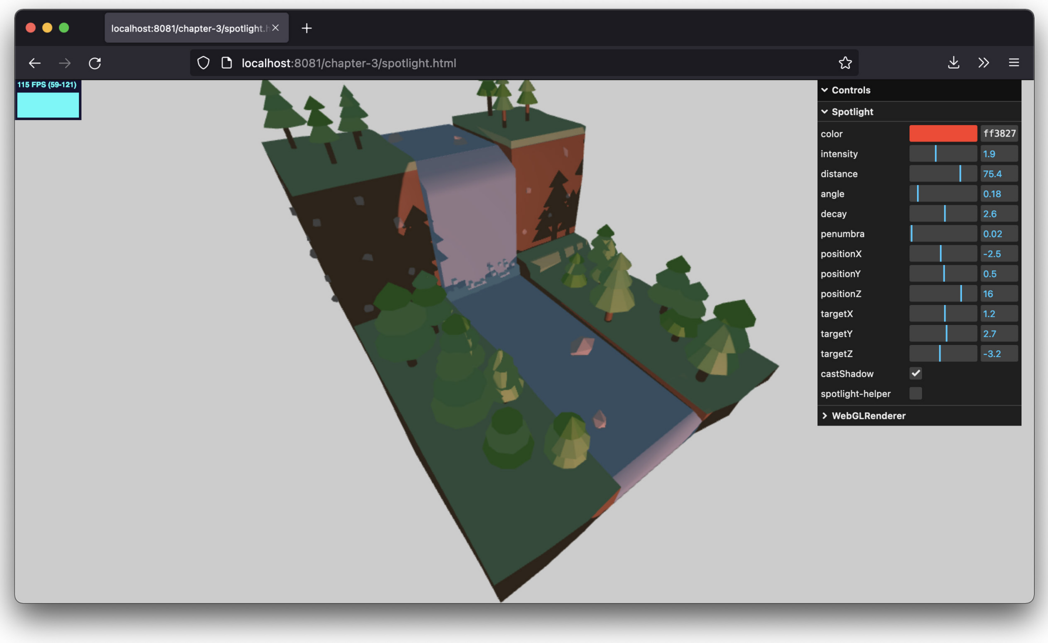
Creating `THREE.SpotLight` is very easy. Just specify the color, set the properties you want, and add it to the scene, as follows:

```
const spotLight = new THREE.SpotLight("#ffffff")
spotLight.penumbra = 0.4;
spotLight.position.set(10, 14, 5);
spotLight.castShadow = true;
spotLight.intensity = 1;
spotLight.shadow.camera.near = 10;
spotLight.shadow.camera.far = 25;
spotLight.shadow.mapSize.width = 2048;
spotLight.shadow.mapSize.height = 2048;
spotLight.shadow.bias = -0.01;
scene.add(spotLight.target);
```

Here, we create an instance of `THREE.SpotLight` and set the various properties to configure the light. We also explicitly set the `castShadow` property to `true` because we want shadows. We also need to point `THREE.SpotLight` somewhere, which we do with the `target` property. Before we can use this property, we first need to add the default `target` of the light to the scene, as follows:

```
scene.add(spotLight.target);
```

By default, the target will be set to `(0, 0, 0)`. In the example for this section, you can change the location of the `target` property and see that the light follows the position of this object:


localhost:8081/chapter-3/spotlight.h
localhost:8081/chapter-3/spotlight.html
115 FPS (59-121)
Controls
Spotlight
WebGLRenderer

color: ff3827
intensity: 1.9
distance: 75.4
angle: 0.18
decay: 2.6
penumbra: 0.02
positionX: -2.5
positionY: 0.5
positionZ: 16
targetX: 1.2
targetY: 2.7
targetZ: -3.2
castShadow ☑
spotlight-helper ☐

Figure 3.8 – Spotlight pointing to a target

Note that you can also set the target of the light to an object in the scene. In that case, the direction of the light will be pointed to that object. If the object that the light is pointed at moves around, the light will keep pointing at that object.

In the table at the beginning of this section, we showed a couple of properties that can be used to control how the light emanates from `THREE.SpotLight`. The `distance` and `angle` properties define the shape of the cone of light. The `angle` property defines the width of the cone, and with the `distance` property, we set the length of the cone. The following diagram explains how these two values define the area that will receive light from `THREE.SpotLight`:

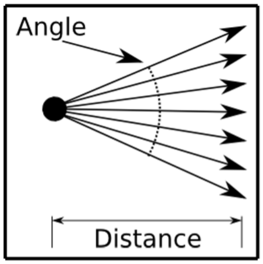
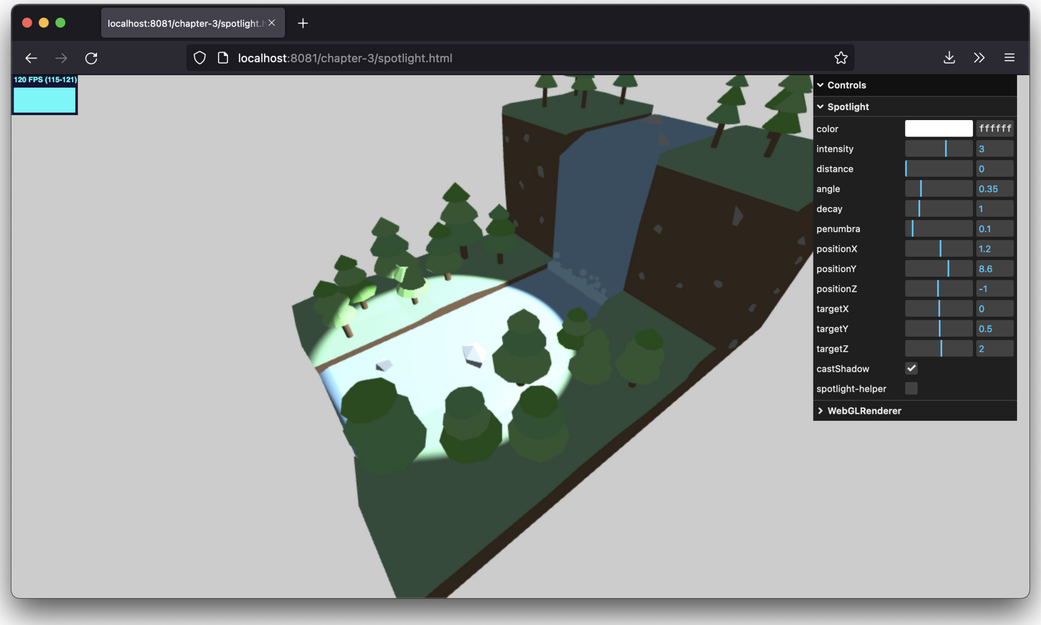

Angle
Distance

Figure 3.9 – Spotlight angle and distance

Usually, you won't need to set these values since they come with reasonable defaults, but you can use these properties, for instance, to create a `THREE.SpotLight` instance that has a very narrow beam or quickly decreases in light intensity. The last property you can use to change the way `THREE.SpotLight` produces light is the `penumbra` property. With this property, you set from what position the intensity of the light decreases at the edge of the light cone. In the following screenshot, you can see the result of the `penumbra` property in action. We have a very bright light (high intensity) that rapidly decreases in intensity as it reaches the edge of the cone:


localhost:8081/chapter-3/spotlight.h
localhost:8081/chapter-3/spotlight.html
120 FPS (115-121)
Controls
Spotlight
WebGLRenderer

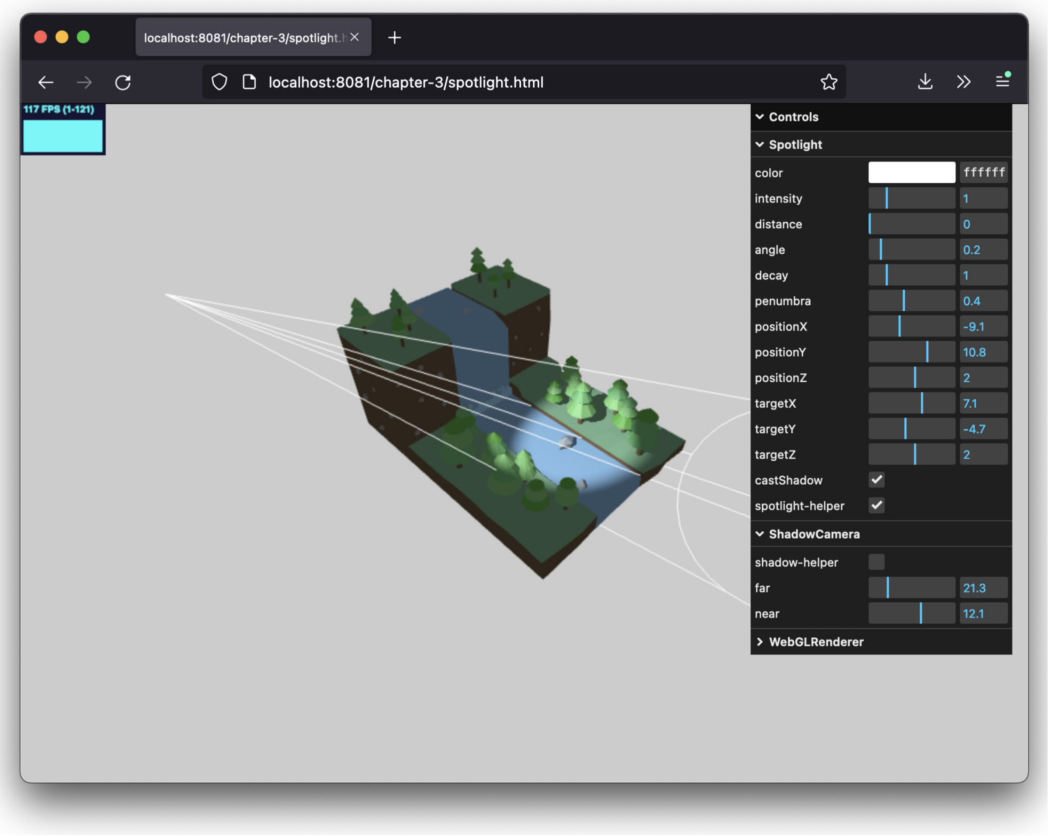
color ffffff
intensity 3
distance 0
angle 0.35
decay 1
penumbra 0.1
positionX 1.2
positionY 8.6
positionZ -1
targetX 0
targetY 0.5
targetZ 2
castShadow ☑
spotlight-helper ☐

Figure 3.10 – Spotlight with a hard penumbra

Sometimes, it can be difficult to determine the correct settings for your lights, just by looking at the rendered scene. You might want to finetune the area that is lit for performance reasons or try and move the light around to a very specific location. This can be achieved by using `THREE.SpotLightHelper`:

```
const spotLightHelper = new THREE.SpotLightHelper
   (spotLight);
scene.add(spotLightHelper)
// in the render loop
spotLightHelper.update();
```

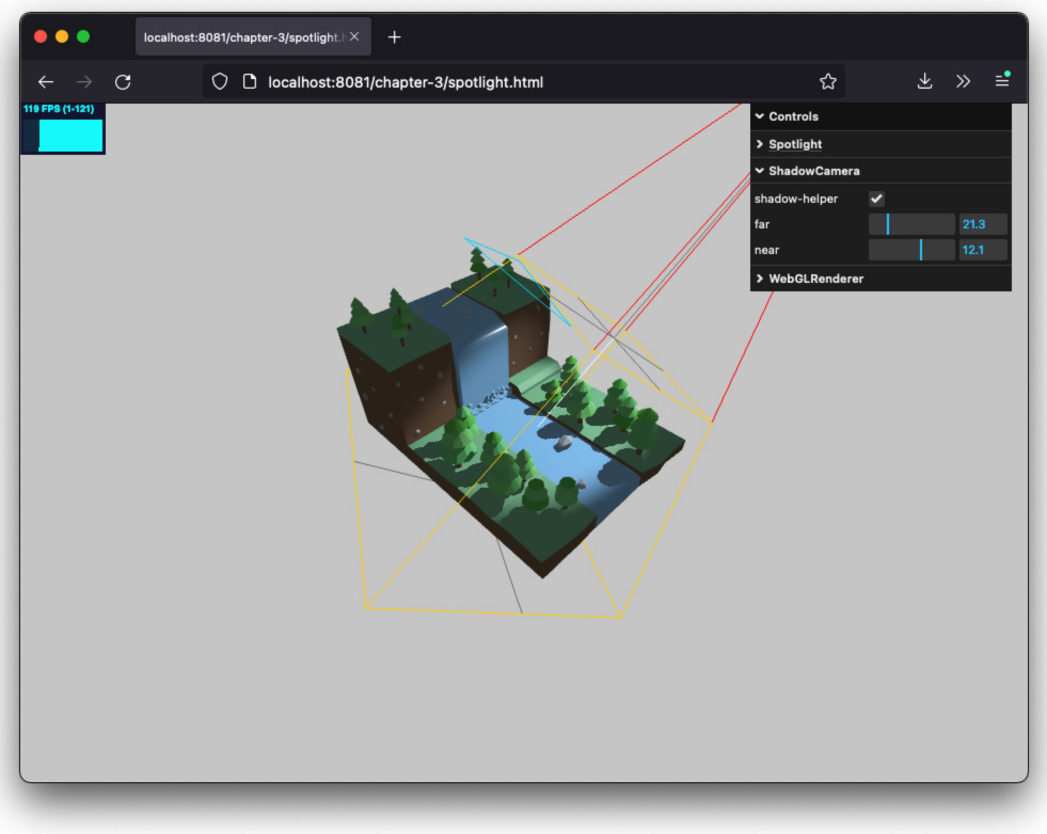
With the preceding code, you get an outline that shows the details of the spotlight, and can help in debugging and correctly positioning and configuring your light:


localhost:8081/chapter-3/spotlight.h
localhost:8081/chapter-3/spotlight.html
117 FPS (1-121)
Controls
Spotlight
ShadowCamera
WebGLRenderer

color ffffff
intensity 1
distance 0
angle 0.2
decay 1
penumbra 0.4
positionX -9.1
positionY 10.8
positionZ 2
targetX 7.1
targetY -4.7
targetZ 2
castShadow ☑
spotlight-helper ☑
shadow-helper ☐
far 21.3
near 12.1

Figure 3.11– Spotlight with the helper enabled

Before moving on to the next light source, we'll quickly look at the shadow-related properties available for a `THREE.SpotLight` object. You've already learned that we can get shadows by setting the `castShadow` property of a `THREE.SpotLight` instance to `true`. You also know that a `THREE.Mesh` object has two shadow-related properties. You set the `castShadow` property for objects that should cast shadows, and you use the `receiveShadow` property for objects that should show a shadow. Three.js also allows you very fine-grained control over how the shadow is rendered. This is done by a couple of the properties we explained in the table at the beginning of this section. With `shadow.camera.near`, `shadow.camera.far`, and `shadow.camera.fov`, you can control how and where this light casts a shadow. For a `THREE.SpotLight` instance, you can't set `shadow.camera.fov` directly. This property is based on the `angle` property of `THREE.SpotLight`. This works in the same way as the perspective camera's field of view, which we explained in *Chapter 2*. The easiest way to see this in action is by adding a `THREE.CameraHelper`; you can do this by checking the menu's `shadow-helper` checkbox and playing around with the camera settings. As you can see in the following screenshot, selecting this checkbox shows the area that is used to determine the shadows for this light:


localhost:8081/chapter-3/spotlight.h
localhost:8081/chapter-3/spotlight.html
119 FPS (1-121)
Controls
Spotlight
ShadowCamera
WebGLRenderer

shadow-helper ☑
far 21.3
near 12.1

Figure 3.12 – Spotlight with the shadow helper enabled

When debugging issues with shadows, adding `THREE.CameraHelper` is useful. To do this, just add the following lines:

```
const shadowCameraHelper = new THREE.CameraHelper
  (spotLight.shadow.camera);
scene.add(shadowCameraHelper);
// in the render loop
shadowCameraHelper.update();
```

I'll end this section with a couple of pointers, just in case you run into issues with shadows.

If the shadow looks blocky, you can increase the `shadow.mapSize.width` and `shadow.mapSize.Height` properties and make sure the area that is used to calculate the shadow tightly wraps your object. You can use the `shadow.camera.near`, `shadow.camera.far`, and `shadow.camera.fov` properties to configure this area.

Remember that you not only have to tell the light to cast shadows but also have to tell each geometry whether it will receive and/or cast shadows by setting the `castShadow` and `receiveShadow` properties.

Shadow bias

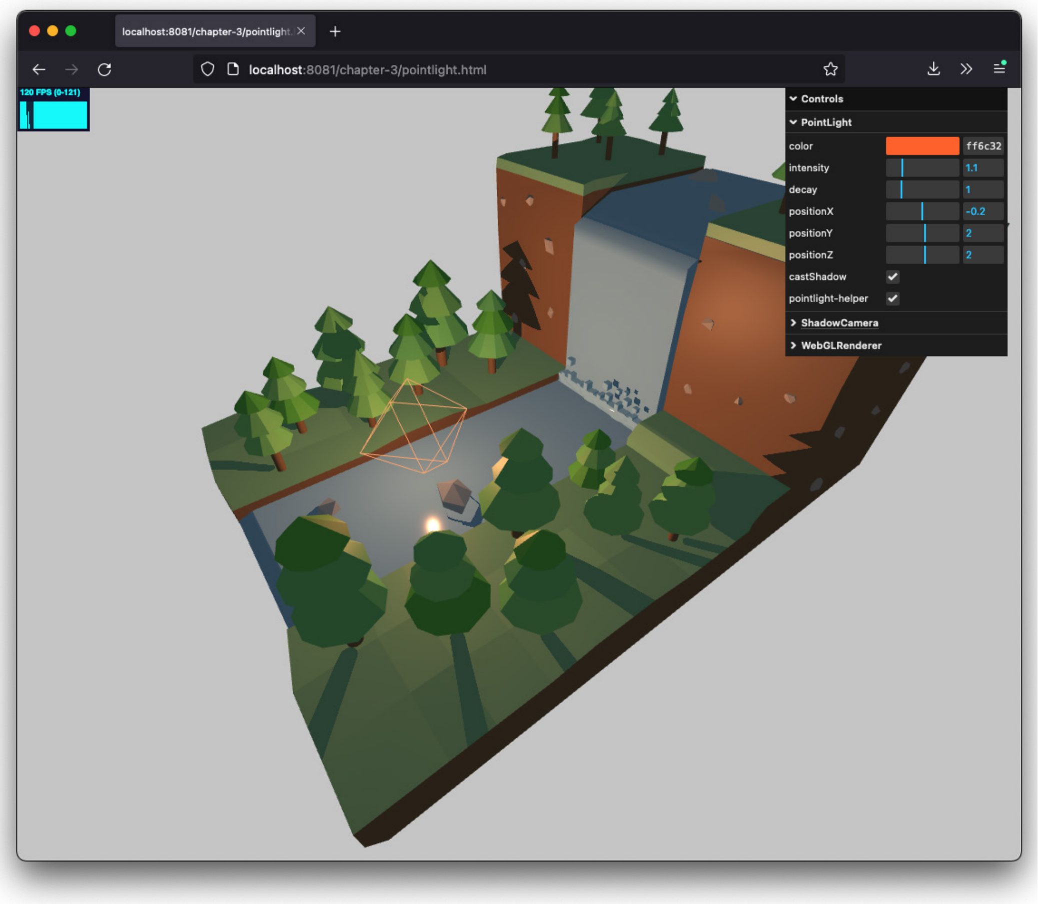
If you use thin objects in your scene, you might see strange artifacts when you render shadows. You can use the `shadow.bias` property to slightly offset the shadows, which will often fix these issues.

If you want to have softer shadows, you can set a different `shadowMapType` value on `THREE.WebGLRenderer`. By default, this property is set to `THREE.PCFShadowMap`; if you set this property to `PCFSoftShadowMap`, you'll get softer shadows.

Now, let's look at the next light source in the list: `THREE.PointLight`.

THREE.PointLight

`THREE.PointLight` is a light source that shines light in all directions emanating from a single point. A good example of a point light is a signal flare fired into the night sky or a campfire. Just as with all the lights, we have a specific example you can use to play around with `THREE.PointLight`. If you look at `point-light.html` in the `chapter-03` folder, you can find an example where a `THREE.PointLight` is being used in the same scene we're also using for the other lights:


localhost:8081/chapter-3/pointlight.
localhost:8081/chapter-3/pointlight.html
120 FPS (0-121)
Controls
PointLight
ShadowCamera
WebGLRenderer

color ff6c32
intensity 1.1
decay 1
positionX -0.2
positionY 2
positionZ 2
castShadow ☑
pointlight-helper ☑

Figure 3.13 – PointLight with the helper enabled

As you can see from the previous screenshot, this light emits to all sides. Just like the spotlight we saw earlier, this light also has a helper, which you can use in the same way. You can see this as the wireframe in the center of the scene:

```
const pointLightHelper = new THREE.PointLightHelper
   (pointLight);
scene.add(pointLightHelper)
// in the render loop
pointLightHelper.update();
```

`THREE.PointLight` shares several properties with `THREE.SpotLight`, which you can use to configure how this light behaves:

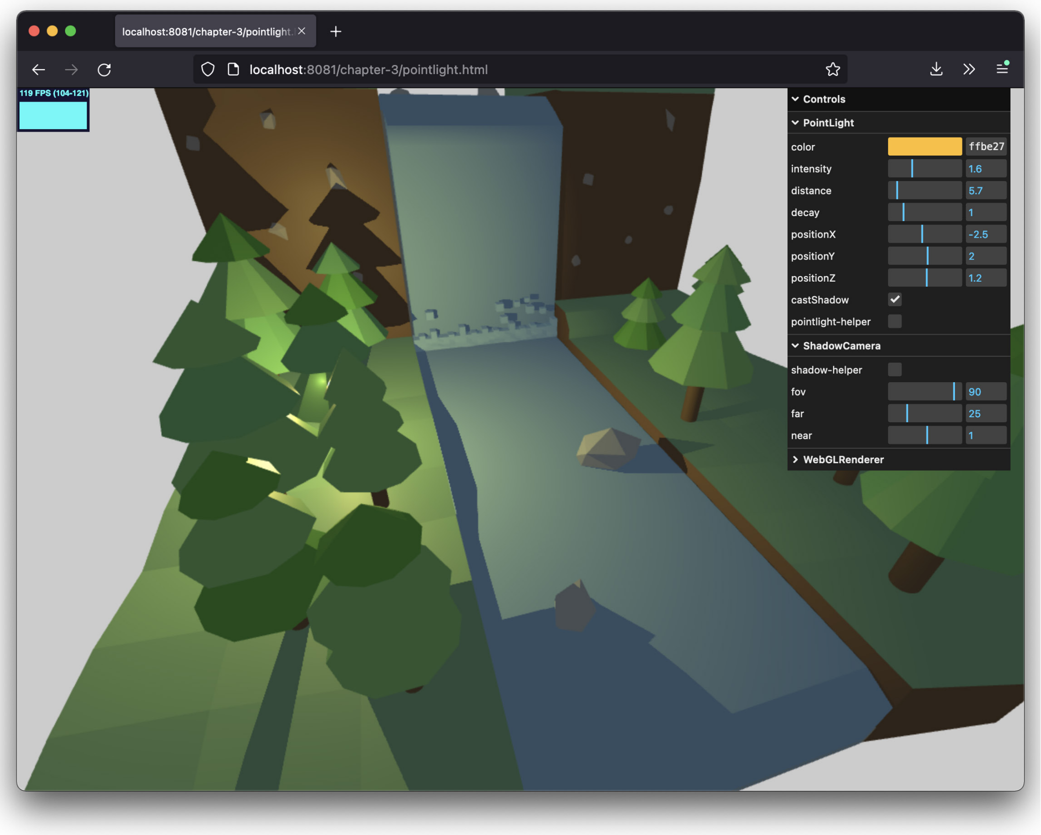
Name	**Description**
`color`	The color of the light that this light source emits.
`distance`	Indicates the distance for which the light shines. The default value is `0`, which means that the light's intensity doesn't decrease based on distance.
`intensity`	Indicates the intensity the light shines with. This defaults to `1`.
`position`	Indicates the position of the light in `THREE.Scene`.
`visible`	Determines whether the light is turned off or on. If this property is set to `true` (the default), this light is turned on, and if set it to `false`, the light is turned off.
`decay`	Indicates the amount the light intensity diminishes the farther you move away from the light source. A `decay` of `2` leads to more realistic light, and the default value is `1`. This property is effective only when the `physicallyCorrectLights` property is set on `WebGLRenderer`.
`power`	Refers to the light's power when being rendered in the physically correct mode (enable this by setting the `physicallyCorrectLights` property set on `WebGLRenderer`). This property is measured in lumens and the default value is `4*Math.PI`. `Power` is also directly related to the `intensity` property (power = intensity * 4π).

Figure 3.14 – Properties of the THREE.PointLight object

Besides these properties, the `THREE.PointLight` object's shadow can be configured in the same way as the shadow for `THREE.SpotLight`. In the next couple of examples and screenshots, we'll show how these properties work for `THREE.PointLight`. First, let's look at how you can create a `THREE.PointLight`:

```
const pointLight = new THREE.PointLight();
scene.add(pointLight);
```

There's nothing special here – we just define the light and add it to the scene; you can, of course, set any of the properties we just showed as well. The two main properties of the `THREE.SpotLight` object are `distance` and `intensity`. With `distance`, you can specify how far the light is emitted before it decays to `0`. For example, in the following screenshot, we set the `distance` property to a low value, and increased the `intensity` property a bit to simulate a campfire between the trees:


localhost:8081/chapter-3/pointlight.
localhost:8081/chapter-3/pointlight.html
119 FPS (104-121)

Controls

PointLight

color ffbe27
intensity 1.6
distance 5.7
decay 1
positionX -2.5
positionY 2
positionZ 1.2
castShadow ☑
pointlight-helper ☐

ShadowCamera

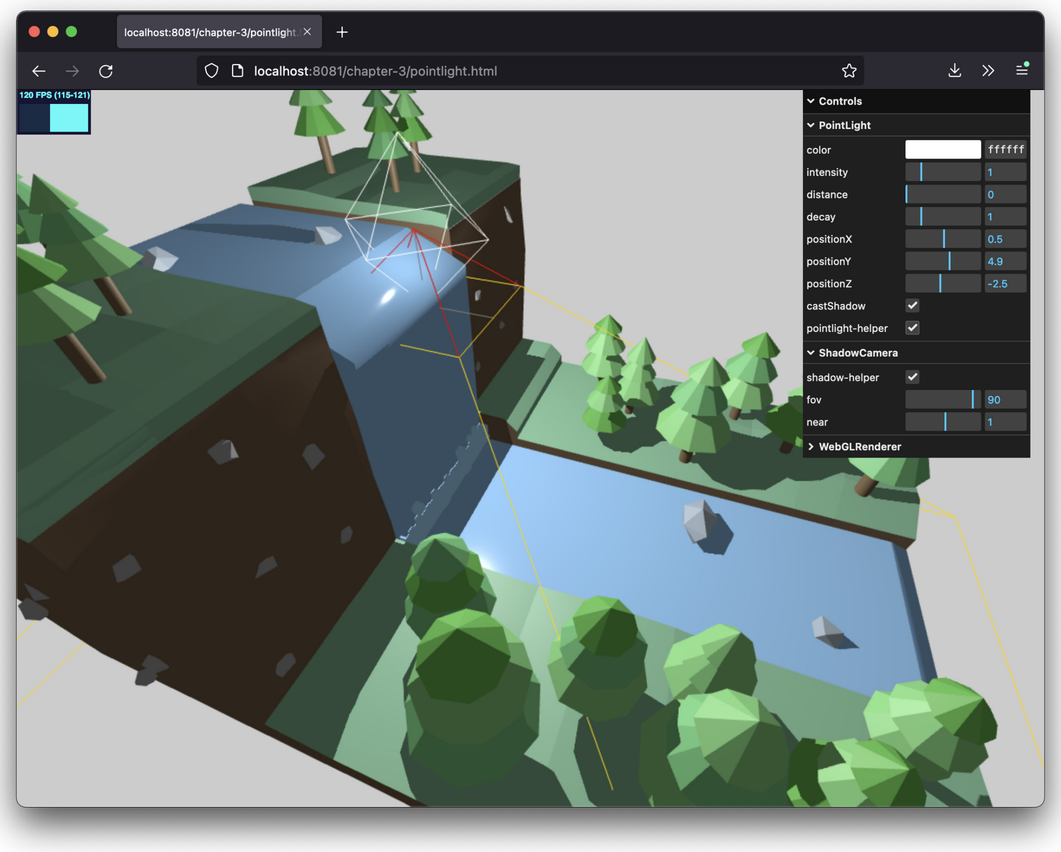
shadow-helper ☐
fov 90
far 25
near 1

WebGLRenderer

Figure 3.15 – PointLight with low distance and higher intensity

You can't set the `power` and decay properties in this example; these properties are really useful if you want to simulate real-world scenarios. A good example of this can be found on the Three.js website: `https://threejs.org/examples/#webgl_lights_physical`.

`THREE.PointLight` also uses a camera to determine where to draw the shadows, so you can use `THREE.CameraHelper` to show what part is covered by that camera. In addition, `THREE.PointLight` provides a helper, `THREE.PointLightHelper`, to show where `THREE.PointLight` shines its light. With both enabled, you get the following very useful debug information:


localhost:8081/chapter-3/pointlight
localhost:8081/chapter-3/pointlight.html
120 FPS (115-121)

Controls

PointLight

color ffffff
intensity 1
distance 0
decay 1
positionX 0.5
positionY 4.9
positionZ -2.5
castShadow ☑
pointlight-helper ☑

ShadowCamera

shadow-helper ☑
fov 90
near 1

WebGLRenderer

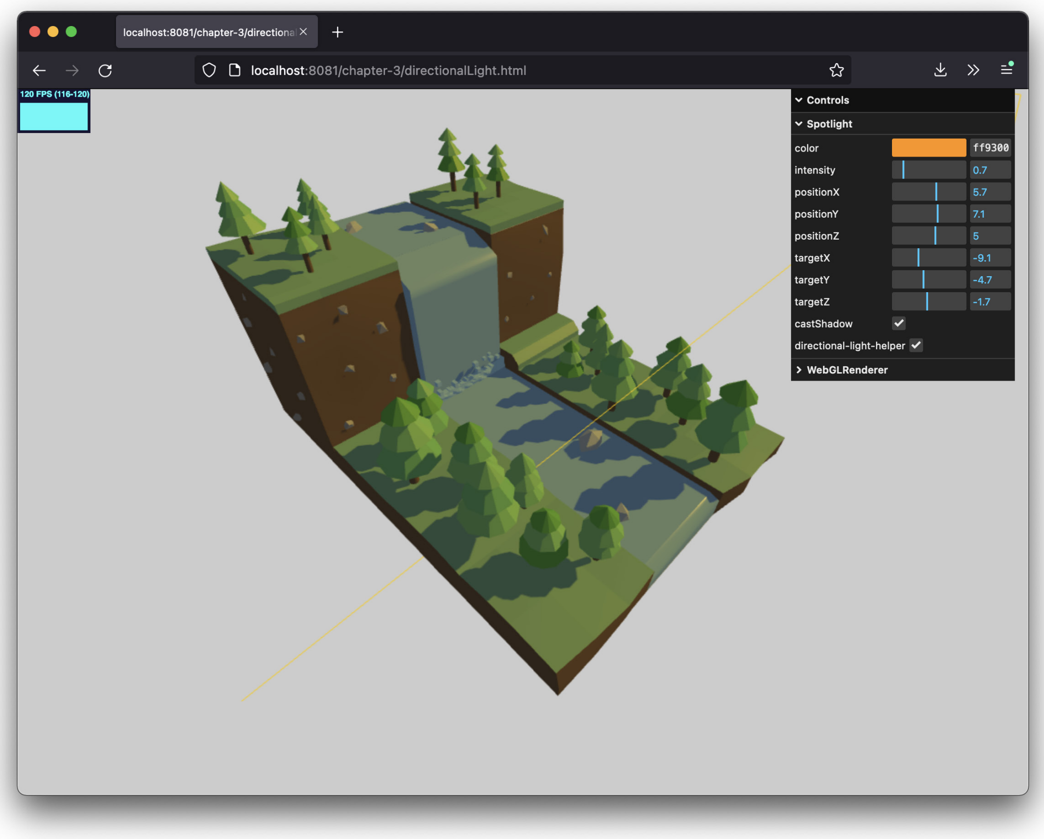
Figure 3.16 – PointLight with helpers enabled

If you look closely at the previous screenshot (*Figure 3.16*), you might notice that shadows are created outside the area the shadow camera is showing. This is because the shadow helper only shows the shadows being cast down from the position of the point light. You can visualize a `THREE.PointLight` as a cube, where each side emits light and can cast shadows. In this case, `THREE.ShadowCameraHelper` only shows the shadows being cast down.

The last of the basic lights that we'll be discussing is `THREE.DirectionalLight`.

THREE.DirectionalLight

This type of light source can be considered as a light that is very far away. All the light rays it sends out are parallel to each other. A good example of this is the Sun. The Sun is so far away that the light rays we receive on Earth are (almost) parallel to each other. The main difference between `THREE.DirectionalLight` and `THREE.SpotLight` (which we saw previously) is that this light won't diminish the farther it gets from the source as it does with `THREE.SpotLight` (you can fine-tune this with the `distance` and `exponent` parameters). The complete area that is lit by `THREE.DirectionalLight` receives the same intensity of light. To see this in action, look at the following `directional-light.html` example:

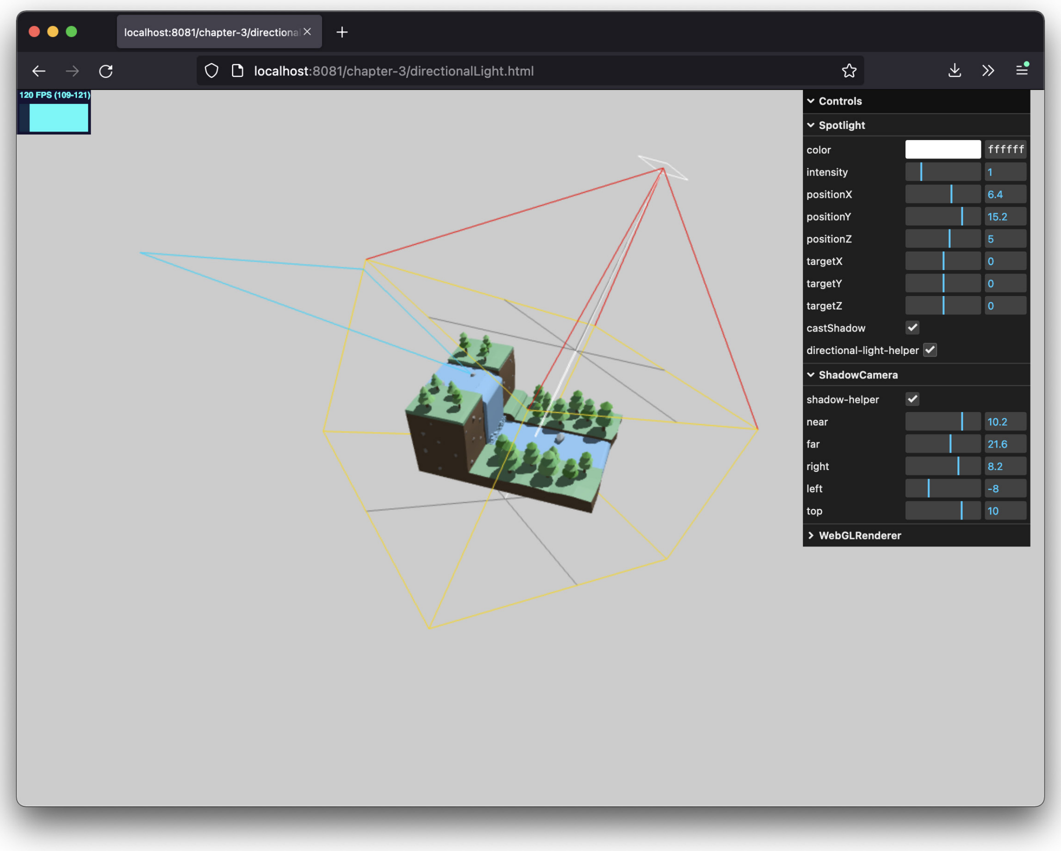

localhost:8081/chapter-3/directiona
localhost:8081/chapter-3/directionalLight.html
120 FPS (116-120)
Controls
Spotlight
WebGLRenderer

color ff9300
intensity 0.7
positionX 5.7
positionY 7.1
positionZ 5
targetX -9.1
targetY -4.7
targetZ -1.7
castShadow ✓
directional-light-helper ✓

Figure 3.17 – Directional light simulating a sunset

As you can see, it is very easy to simulate, for instance, a sunset, using a `THREE.DirectionalLight`. Just as with `THREE.SpotLight`, there are a couple of properties you can set on this light. For example, you can set the `intensity` property of the light and the way it casts shadows. `THREE.DirectionalLight` has a lot of properties that are the same as those of `THREE.SpotLight`: `position`, `target`, `intensity`, `castShadow`, `shadow.camera.near`, `shadow.camera.far`, `shadow.mapSize.width`, `shadow.mapSize.width`, and `shadowBias`. For more information on those properties, you can look at the preceding section on `THREE.SpotLight`.

If you look back at the `THREE.SpotLight` examples, you will see that we had to define the cone of light where shadows were applied. Since all the rays are parallel to each other for `THREE.DirectionalLight`, we don't have a cone where shadows need to be applied; instead, we have a cuboid area (represented internally with a `THREE.OrthographicCamera`), as you can see in the following screenshot, where we enabled the shadow helper:


localhost:8081/chapter-3/directiona
localhost:8081/chapter-3/directionalLight.html
120 FPS (109-121)

Controls

Spotlight

color ffffff
intensity 1
positionX 6.4
positionY 15.2
positionZ 5
targetX 0
targetY 0
targetZ 0
castShadow ☑
directional-light-helper ☑

ShadowCamera

shadow-helper ☑
near 10.2
far 21.6
right 8.2
left -8
top 10

WebGLRenderer

Figure 3.18 – Directional light showing a cuboid shadow area

Everything that falls within this cube can cast and receive shadows from the light. Just as for `THREE.SpotLight`, the tighter you define this area around the objects, the better your shadows will look. Define this cube using the following properties:

```
directionalLight.castShadow = true;
directionalLight.shadow.camera.near = 2;
directionalLight.shadow.camera.far = 80;
directionalLight.shadow.camera.left = -30;
directionalLight.shadow.camera.right = 30;
directionalLight.shadow.camera.top = 30;
directionalLight.shadow.camera.bottom = -30;
```

You can compare this with the way we configured the orthographic camera in the *Using different cameras for different scenes* section in *Chapter 2*.

As we've already seen in this section, light sources use colors. For now, we've just configured the colors using a hex string, but the `THREE.Color` object provides a lot of different options for creating the initial color object. In this next section, we'll explore the functionality provided by the `THREE.Color` object.

Using the THREE.Color object

In Three.js, when you need to provide a color (for example, for materials, lights, and so on), you can pass in a `THREE.Color` object; otherwise, Three.js will create one from a passed-in string value, as we saw for `THREE.AmbientLight`. Three.js is very flexible when parsing the input for the `THREE.Color` constructor. You can create a `THREE.Color` object in the following ways:

- **Hex string**: `new THREE.Color("#ababab")` will create a color based on the passed-in CSS color string.
- **Hex value**: `new THREE.Color(0xababab)` will create the color based on the passed-in hex value. If you know the hex value, this is usually the best approach.
- **RGB string**: `new THREE.Color("rgb(255, 0, 0)")` or `new THREE.Color("rgb(100%, 0%, 0%)"`.
- **Color name**: You can use named colors as well – for example, `new THREE.Color( 'skyblue' )`.
- **HSL string**: If you like working in the HSL domain instead of the RGB domain, you can pass in the HSL values with `new THREE.Color("hsl(0, 100%, 50%)")`.
- **Separate RGB values**: You can specify the individual RGB components on a scale from 0 to 1: `new THREE.Color( 1, 0, 0 )`.

If you want to change the color after construction, you'll have to create a new `THREE.Color` object or modify the internal properties of the `THREE.Color` object. The `THREE.Color` object comes with a large set of properties and functions. The first group of functions allows you to set the color of a `THREE.Color` object:

- `set(value)`: Sets the value of a color to the supplied hex value. This hex value may be a string, a number, or an existing `THREE.Color` instance.

- `setHex(value)`: Sets the value of a color to the supplied numeric hex value.
- `setRGB(r,g,b)`: Sets the value of a color based on the supplied RGB values. The values range from `0` to `1`.
- `setHSL(h,s,l)`: Sets the value of this color on the supplied HSL values. The values range from 0 to 1. A good explanation of how HSL works for configuring colors can be found at `http://en.wikibooks.org/wiki/Color_Models:_RGB,_HSV,_HSL`.
- `setStyle(style)`: Sets the value of a color based on the CSS way of specifying colors. For instance, you could use `rgb(255,0,0)`, `#ff0000`, `#f00`, or even `red`.

If you've already got an existing `THREE.Color` instance and want to use that color, you can use the following functions:

- `copy(color)`: Copies the color values from the `THREE.Color` instance provided to this color.
- `copySRGBToLinear(color)`: Sets the color of this object based on the `THREE.Color` instance supplied. The color is first converted from the sRGB color space into the linear color space. The sRGB color space uses an exponential scale instead of a linear one. More information on the sRGB color space can be found here: `https://www.w3.org/Graphics/Color/sRGB.html`.
- `copyLinearToSRGB(color)`: Sets the color of this object based on the `THREE.Color` instance supplied. The color is first converted from the linear color space into the sRGB color space.
- `convertSGRBToLinear()`: Converts the current color from the sRGB color space into the linear color space.
- `convertLinearToSGRB()`: Converts the current color from the linear color space into the sRGB color space.

If you want information on the currently configured color, the `THREE.Color` object also provides some helper functions for that:

- `getHex()`: Returns the value from this color object as a number: `435241`.
- `getHexString()`: Returns the value from this color object as a hex string: `0c0c0c`.
- `getStyle()`: Returns the value from this color object as a CSS-based value: `rgb(112,0,0)`.
- `getHSL(target)`: Returns the value from this color object as an HSL value (`{ h: 0, s: 0, l: 0 }`). If you provide the optional `target` object, Three.js will set the `h`, `s`, and `l` properties on that object.

Three.js also provides functions to change the current color by modifying the individual color components. This is shown here:

- `offsetHSL(h, s, l)`: Adds the `h`, `s`, and `l` values provided to the `h`, `s`, and `l` values of the current color.

- `add(color)`: Adds the `r`, `g`, and `b` values of the color supplied to the current color.
- `addColors(color1, color2)`: Adds `color1` and `color2` and sets the value of the current color to the result.
- `addScalar(s)`: Adds a value to the RGB components of the current color. Bear in mind that the internal values use a range from `0` to `1`.
- `multiply(color)`: Multiplies the current RGB values with the RGB values from `THREE.Color`.
- `multiplyScalar(s)`: Multiplies the current RGB values with the value supplied. Remember that the internal values range between `0` to `1`.
- `lerp(color, alpha)`: Finds the color that is between the color of this object and the `color` property supplied. The `alpha` property defines how far between the current color and the supplied color you want the result to be.

Finally, there are a couple of basic helper methods available:

- `equals(color)`: Returns `true` if the RGB values of the `THREE.Color` instance supplied match the values of the current color
- `fromArray(array)`: Has the same functionality as `setRGB`, but now, the RGB values can be provided as an array of numbers
- `toArray`: Returns an array with three elements: `[r, g, b]`
- `clone`: Creates an exact copy of a color

In the preceding lists, you can see that there are many ways in which you can change the current color. A lot of these functions are used internally by Three.js, but they also provide a good way to easily change the color of lights and materials, without having to create and assign new `THREE.Color` objects.

So far, we've looked at the basic lights provided by Three.js and how shadows work. In most cases, you'll use a combination of these lights for your scene. Three.js also provides a couple of special lights for very specific use cases. We'll look at those in the next section.

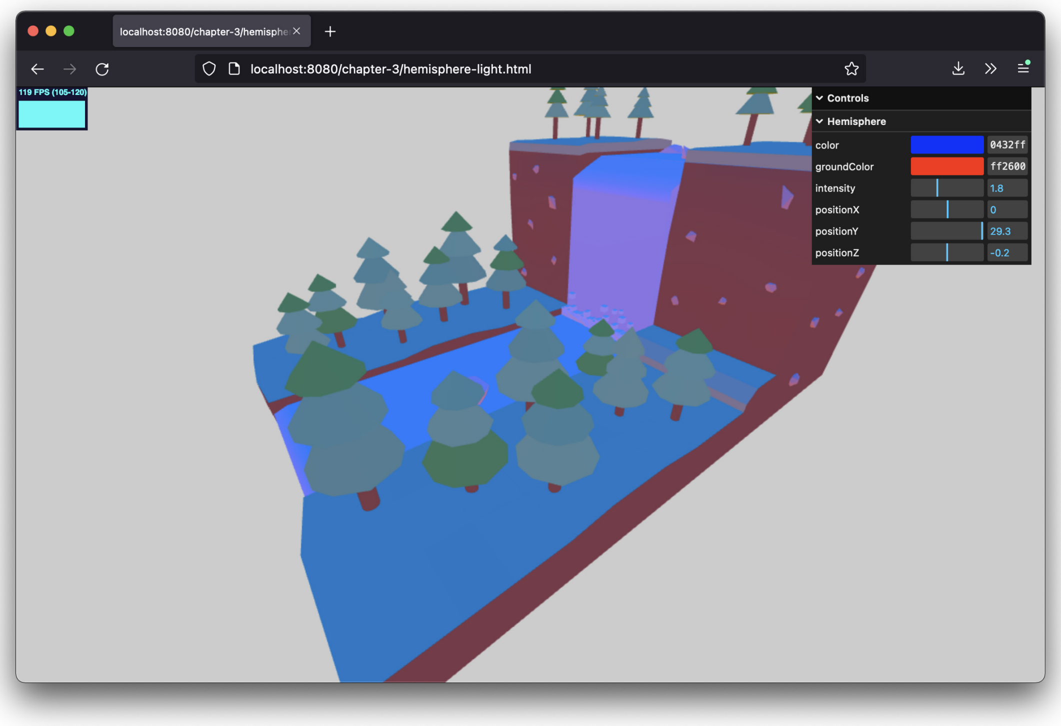
Working with special lights

In this section on special lights, we'll discuss three additional lights provided by Three.js. First, we'll discuss `THREE.HemisphereLight`, which helps in creating more natural lighting for outdoor scenes. Then, we'll look at `THREE.RectAreaLight`, which emits lights from a large area instead of a single point. Next, we'll look at how we can use a `LightProbe` to apply light based on a cubemap, and finally, we'll show you how you can add a lens flare effect to your scene.

The first special light we're going to look at is `THREE.HemisphereLight`.

THREE.HemisphereLight

With `THREE.HemisphereLight`, we can create more natural-looking outdoor lighting. Without this light, we could simulate the outdoors by creating `THREE.DirectionalLight`, which emulates the sun, and maybe add another `THREE.AmbientLight` to provide some general color to the scene. However, doing so won't look natural. When you're outdoors, not all the light comes directly from above: much is diffused by the atmosphere and reflected by the ground and other objects. `THREE.HemisphereLight` in Three.js was created for this scenario. This is an easy way to get more natural-looking outdoor lighting. To see an example, look at `hemisphere-light.html` in the following figure:


localhost:8080/chapter-3/hemisphe
localhost:8080/chapter-3/hemisphere-light.html
119 FPS (105-120)
Controls
Hemisphere

color 0432ff
groundColor ff2600
intensity 1.8
positionX 0
positionY 29.3
positionZ -0.2

Figure 3.19 – Hemisphere light

If you look closely at this screenshot, you will see that the ground color of the hemisphere is shown more at the bottom of the sphere, and the sky color (set through the `color` property) is visible at the top of the scene. In this example, you can set these colors and their intensity. Creating a hemisphere light is just as easy as creating any of the other lights:

```
const hemiLight = new THREE.HemisphereLight(0x0000ff,
  0x00ff00, 0.6); hemiLight.position.set(0, 500, 0);
scene.add(hemiLight);
```

You just specify the color that is received from the sky, the color received from the ground, and the intensity of these lights. If you want to change these values later on, you can access them through the following properties:

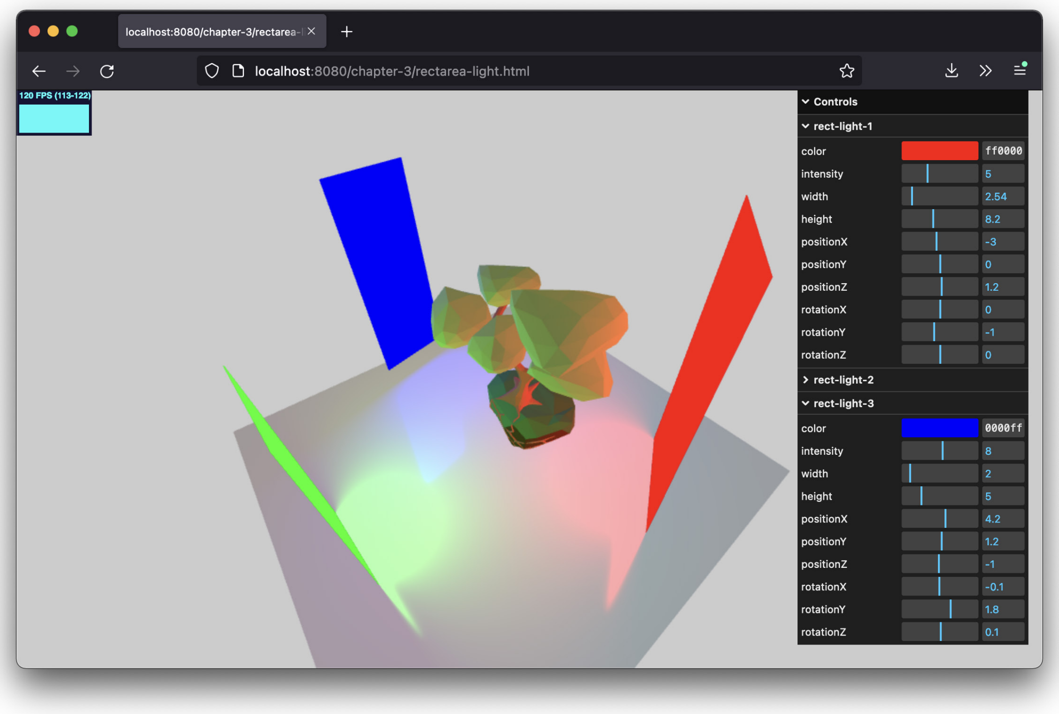
Property	**Description**
`groundColor`	Indicates the color that is emitted from the ground
`color`	Implies the color that is emitted from the sky
`intensity`	Implies the intensity with which the light shines

Figure 3.20 – Properties of the THREE.HemisphereLight object

Since a `HemisphereLight` acts like a `THREE.AmbientLight` object and just adds color to all the objects in the scene, it isn't capable of casting shadows. The lights we've seen so far are more traditional. The next property allows you to simulate light from rectangular light sources – for instance, a window or a computer screen.

THREE.RectAreaLight

With `THREE.RectAreaLight`, we can define a rectangular area that emits light. Before we look at the details, let's first look at the result we're aiming for (`rectarea-light.html` opens this example); the following screenshot shows a couple of `THREE.RectAreaLight` objects:


localhost:8080/chapter-3/rectarea-l
localhost:8080/chapter-3/rectarea-light.html
120 FPS (113-122)

Controls

rect-light-1

color ff0000
intensity 5
width 2.54
height 8.2
positionX -3
positionY 0
positionZ 1.2
rotationX 0
rotationY -1
rotationZ 0

rect-light-2

rect-light-3

color 0000ff
intensity 8
width 2
height 5
positionX 4.2
positionY 1.2
positionZ -1
rotationX -0.1
rotationY 1.8
rotationZ 0.1

Figure 3.21 – RectArea lights emitting across their complete surface

What you see in this screenshot is that we've defined three `THREE.RectAreaLight` objects, each with its own color. You can see how these lights affect the whole area, and when you move them around or change their position, you can see how the different objects in the scene are affected.

We haven't explored the different materials and how light affects them. We'll do this in the next chapter, *Chapter 4*, *Working with Three.js Materials*. A `THREE.RectAreaLight` only works with `THREE.MeshStandardMaterial` or `THREE.MeshPhysicalMaterial`. More on these materials will be presented in *Chapter 4*.

To work with a `THREE.RectAreaLight`, we need to take a couple of small additional steps. First, we need to load and initialize `RectAreaLightUniformsLib`; the following is a set of additional low-level WebGL scripts needed by this light:

```
import { RectAreaLightUniformsLib } from "three/examples
  /jsm/lights/RectAreaLightUniformsLib.js";
...
RectAreaLightUniformsLib.init();
```

Next, we can create the `THREE.AreaLight` object just like any other light:

```
const rectLight1 = new THREE.RectAreaLight
  (0xff0000, 5, 2, 5);
rectLight1.position.set(-3, 0, 5);
scene.add(rectLight1);
```

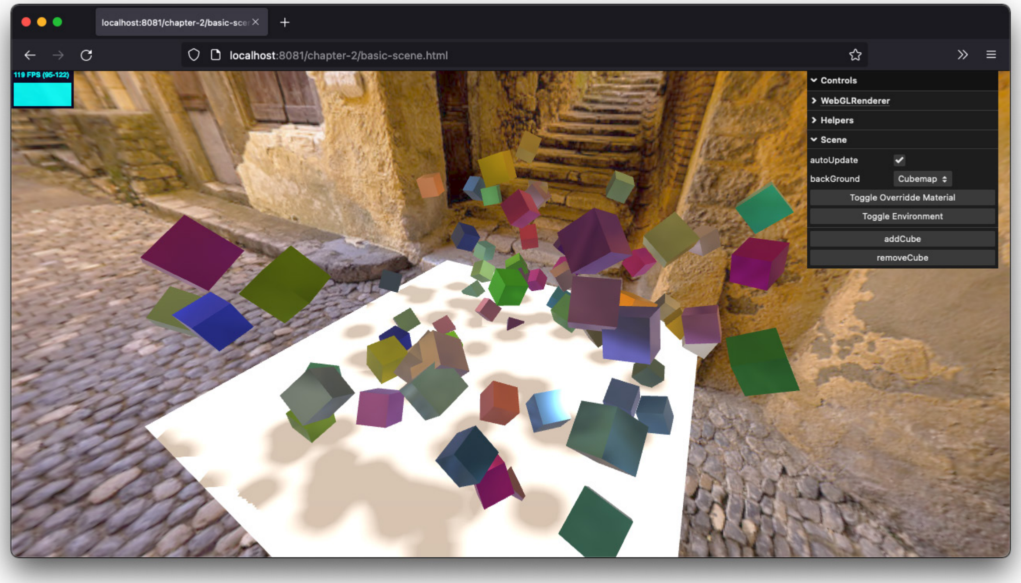
If you look at the constructor of this object, you will see that it takes four properties. The first one is the color of the light, the second one is the intensity, and the last two define how large the area of this light is. Note that if you want to visualize these lights, as we did in the example, you have to create a rectangle yourself at the same position, rotation, and size as your `THREE.RectAreaLight`.

This light can be used to create some nice effects, but it'll probably take some experimenting to get the effect you want. Once again, in this example, you've got a menu on the right-hand side that you can use to play around with the various settings.

In recent versions of Three.js, a new light was added called `THREE.LightProbe`. This light is similar to `THREE.AmbientLight` but takes the cubemap of `WebGLRenderer` into account. This is the last light source that we'll discuss in this chapter.

THREE.LightProbe

In the previous chapter, we talked a little bit about what a cubemap is. With a cubemap, you can show your models inside an environment. In the previous chapter, we used a cubemap to create a background that rotates with the view of the camera:



localhost:8081/chapter-2/basic-scene.html

autoUpdate ☑

backGround Cubemap

Figure 3.22 – Cubemap example from Chapter 2

As we'll see in the next chapter, we can use the information from a cubemap to show reflections on our materials. Normally, though, these environment maps don't contribute any light to your scene. With a `THREE.LightProbe`, however, we can extract lighting level information from the cubemap and use it to illuminate our models. So, what you'll get looks a bit like a `THREE.AmbientLight`, but it affects the objects based on their location in the scene and the information from the cubemap.

The easiest way to explain this is by looking at an example. Open up `light-probe.html` in your browser; you'll see the following scene:

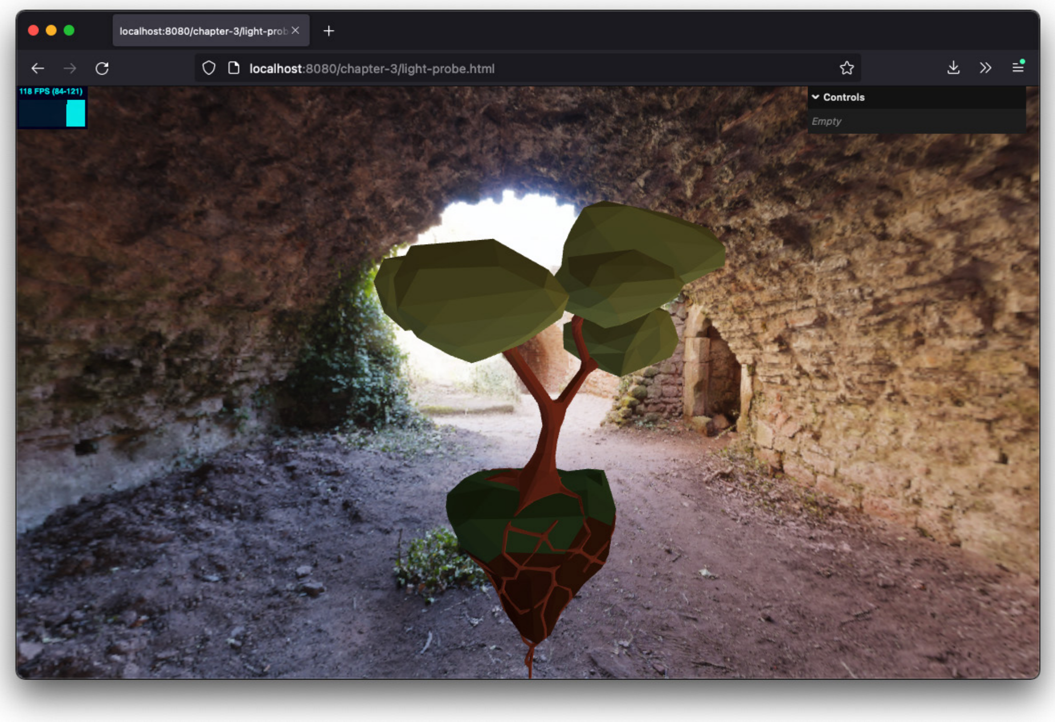

localhost:8080/chapter-3/light-prob
localhost:8080/chapter-3/light-probe.html
118 FPS (84-121)
Controls
Empty

Figure 3.23 – LightProbe with a model in a cave

In the preceding example, we've got a model inside a cave-like environment. What you can see, if you rotate the camera around, is that based on the light of the environment, our model is slightly differently lit. In the previous screenshot, we're looking at the back of the object, which is further down in the cave, so the model is darker on that side. If we completely rotate the camera and set the entrance of the cave to our back, we'll see that the model is much brighter and receives more light:

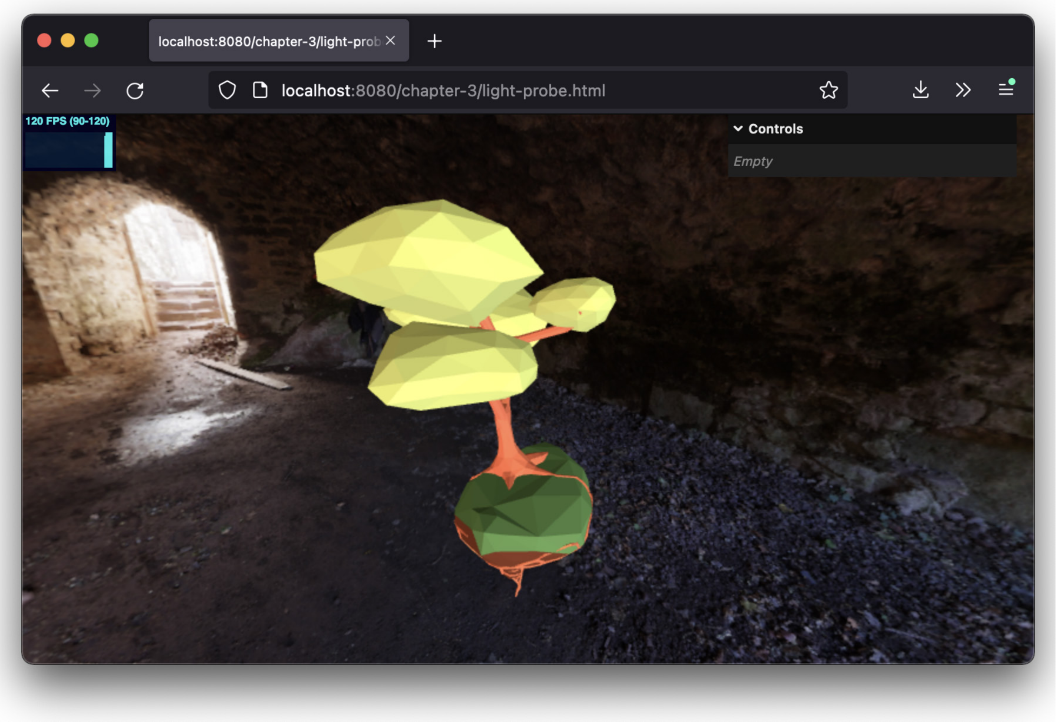

localhost:8080/chapter-3/light-prob
localhost:8080/chapter-3/light-probe.html
120 FPS (90-120)
Controls
Empty

Figure 3.24 – LightProbe with a model in a cave receiving more light

This is a really neat trick to make your objects look more life-like and less flat, and with a `THREE.LightProbe`, your model will receive light non-uniformly, which looks much better.

Setting up a `THREE.LightProbe` is a bit more work but only needs to be done once when you create your scene. So long as you don't change the environment, you don't need to recalculate the values of the `THREE.LightProbe` object:

```
Import { LightProbeGenerator } from "three/examples/
  jsm/lights//LightProbeGenerator";
...
const loadCubeMap = (renderer, scene) => {
  const base = "drachenfels";
  const ext = "png";
  const urls = [
    "/assets/panorama/" + base + "/posx." + ext,
    "/assets/panorama/" + base + "/negx." + ext,
    "/assets/panorama/" + base + "/posy." + ext,
    "/assets/panorama/" + base + "/negy." + ext,
    "/assets/panorama/" + base + "/posz." + ext,
    "/assets/panorama/" + base + "/negz." + ext,
  ];
  new THREE.CubeTextureLoader().load(urls, function
    (cubeTexture) {
    cubeTexture.encoding = THREE.sRGBEncoding;
    scene.background = cubeTexture;
    const lp = LightProbeGenerator.fromCubeTexture
      (cubeTexture);
    lp.intensity = 15;
    scene.add(lp);
  });
};
```

In the preceding code fragment, we do two main things. First, we use `THREE.CubeTextureLoader` to load in a cubemap. As we'll see in the next chapter, a cubemap consists of six images representing the six sides of a cube, which together will make up our environment. Once that has been loaded, we set this to the background of our scene (note that this isn't needed for `THREE.LightProbe` to work).

Now that we've got this cubemap, we can generate a `THREE.LightProbe` from it. This is done by passing in `cubeTexture` to a `LightProbeGenerator`. The result is a `THREE.LightProbe`, which we add to our scene, just like any other light. Just like with a `THREE.AmbientLight`, you can control how much this light contributes to the lighting of your meshes by setting the `intensity` property.

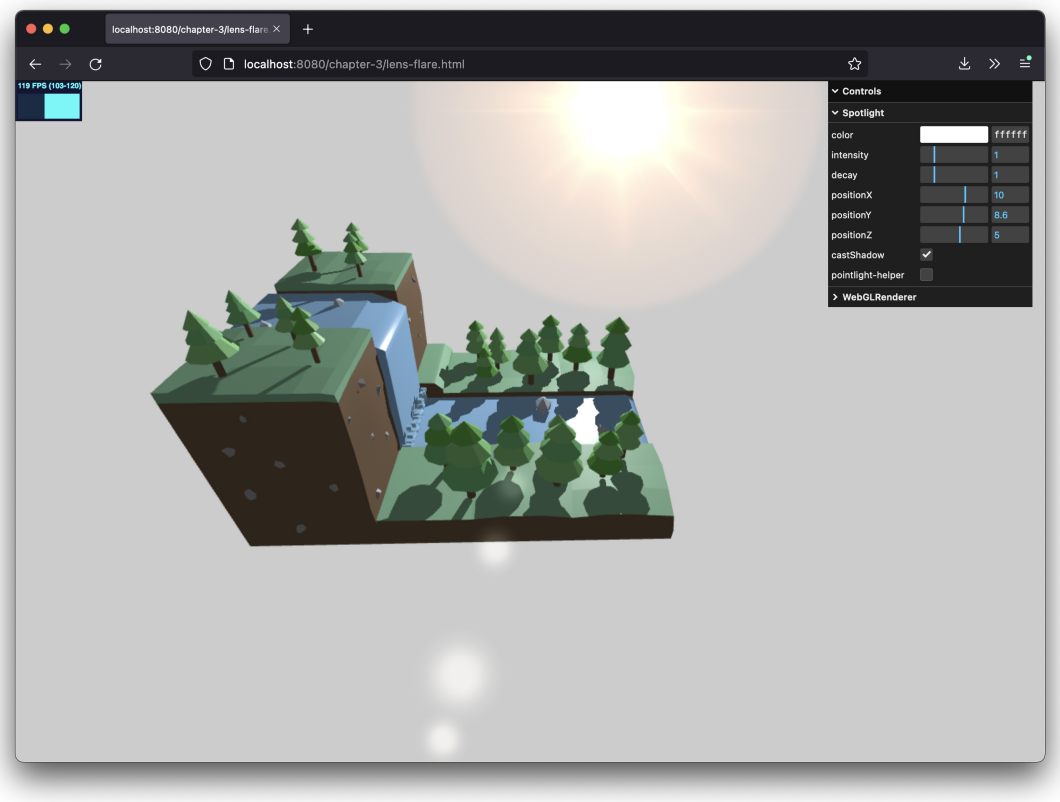
Note

There is also another kind of `LightProbe` provided by Three.js: `THREE.HemisphereLightProbe`. This one works pretty much the same as a normal `THREE.HemisphereLight`, but uses a `LightProbe` internally.

The final object in this chapter isn't a light source but plays a trick on the camera often seen in movies: `THREE.LensFlare`.

THREE.LensFlare

You are probably already familiar with lens flares. For instance, they appear when you take a direct photograph of the Sun or another bright light source. In most cases, you want to avoid this, but for games and 3D-generated images, it provides a nice effect that makes scenes look a bit more realistic. Three.js also has support for lens flares and makes it very easy to add them to your scene. In this last section, we're going to add a lens flare to a scene and create the output shown in the following screenshot; you can see this for yourself by opening `lens-flare.html`:


localhost:8080/chapter-3/lens-flare
localhost:8080/chapter-3/lens-flare.html
119 FPS (103-120)
Controls
Spotlight
WebGLRenderer

color ffffff
intensity 1
decay 1
positionX 10
positionY 8.6
positionZ 5
castShadow ☑
pointlight-helper ☐

Figure 3.25 – A lens flare appears when you look into the light

We can create a lens flare by instantiating the `LensFlare` object and by adding `LensFlareElement` objects:

```
import {
  Lensflare,
  LensflareElement,
} from "three/examples/jsm/objects/Lensflare";
const textureLoader = new THREE.TextureLoader()
const textureFlare0 = textureLoader.load
  ('/assets/textures/lens-flares/lensflare0.png')
const textureFlare1 = textureLoader.load
  ('/assets/textures/lens-flares/lensflare3.png')

const lensFlare = new LensFlare();
```

```
lensFlare.addElement(new LensflareElement
  (textureFlare0, 512, 0));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 60, 0.6));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 70, 0.7));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 120, 0.9));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 70, 1.0));
pointLight.add(lensFlare);
```

The `LensFlare` element is just a container for our `LensFlareElement` objects, and `LensFlareElement` is the artifact you see when you look at the light source. Then, we add `LensFlare` to the light source, and we're done. If you look at the code, you will see that we pass in several properties for each `LensFlareElement`. These properties determine what `LensFlareElement` looks like and where it is rendered on screen. To use this element, we can apply the following constructor arguments:

Property	**Description**
`texture`	A texture is an image that determines the shape of the flare.
`size`	We can specify how large the flare should be. `size` denotes the size in pixels. If you specify `-1`, the size of the texture itself is used.
`distance`	Indicates the distance from the light source (`0`) to the camera (`1`). Use this to position the lens flare in the right position.
`color`	Denotes the color of the flare.

Figure 3.26 – Properties of the THREE.LensFlareElement object

First, let's look a bit closer at the first `LensFlareElement`:

```
const textureLoader = new THREE.TextureLoader();
const textureFlare0 = textureLoader.load(
  "/assets/textures/lens-flares/lensflare0.png"
);
lensFlare.addElement(new LensflareElement
  (textureFlare0, 512, 0));
```

The first argument, `texture`, is an image that shows the shape and some basic coloring of the flare. We load this with a `THREE.TextureLoader`, where we simply add the location of `texture`:

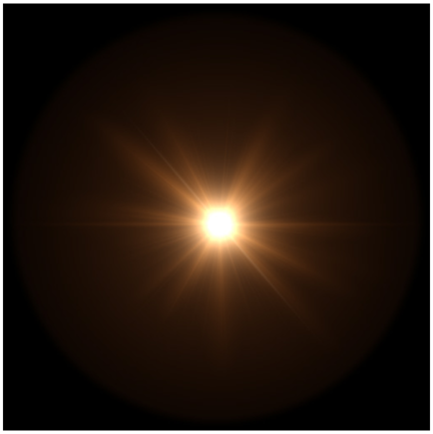


Figure 3.27 – Lens flare used in the example

The second argument is the size of this flare. Since this is the flare we see at the light source itself, we're going to make it quite big: `512` pixels in this case. Next, we need to set the `distance` property of this flare. What you set here is the relative distance between the source of the light and the center of the camera. If we set a distance of `0`, the texture will be shown at the position of the light, and if we set it to `1`, it will be shown at the position of the camera. In this case, we put it directly at the light source.

Now, if you look back at the position of the other `LightFlareElement` objects, you will see that we positioned them at intervals from `0` to `1`, which results in the effect you see when you open up the `lens-flare.html` example:

```
const textureFlare1 = textureLoader.load(
  "/assets/textures/lens-flares/lensflare3.png"
);
lensFlare.addElement(new LensflareElement
  (textureFlare1, 60, 0.6));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 70, 0.7));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 120, 0.9));
lensFlare.addElement(new LensflareElement
  (textureFlare1, 70, 1.0));
```

With that, we've discussed the various lighting options provided by Three.js.

Summary

In this chapter, we covered a lot of information about the different kinds of lights that are available in Three.js. You learned that configuring lights, colors, and shadows is not an exact science. To get the correct result, you should experiment with the different settings and use a `lil.GUI` control to fine-tune your configuration. The different lights behave in different ways and, as we'll see in *Chapter 4*, materials respond differently to lights as well.

A `THREE.AmbientLight` color is added to every color in the scene and is often used to smooth hard colors and shadows. `THREE.PointLight` emits light in all directions and can cast shadows. `THREE.SpotLight` is a light that resembles a flashlight. It has a conical shape, can be configured to fade over distance, and can cast shadows. We also looked at `THREE.DirectionalLight`. This light can be compared to a faraway light, such as the Sun, whose light rays travel parallel to each other, the intensity of which doesn't decrease the farther away it gets from the configured target, and which can also cast shadows.

Besides the standard lights, we also looked at a couple of more specialized lights. For a more natural outdoor effect, you can use `THREE.HemisphereLight`, which takes into account ground and sky reflections. `THREE.RectAreaLight` doesn't shine from a single point but emits light from a large area. We also showed a more advanced sort of ambient lighting by using a `THREE.LightProbe`, which used information from an environment map to determine how an object is lit. Finally, we showed you how to add a photographic lens flare with the `THREE.LenseFlare` object.

In the chapters so far, we have already introduced a couple of different materials, and in this chapter, you saw that not all materials respond in the same manner to the available lights. In *Chapter 4*, we'll provide an overview of the materials that are available in Three.js.

Part 2:
Working with the Three.js Core Components

In this second part, we'll dive into the different materials provided by Three.js and the different geometries you can use to create your own scenes. Besides the geometries, we'll also look at how Three.js supports points and sprites, which you can use, for instance, for rain and smoke effects.

In this part, there are the following chapters:

- *Chapter 4, Working with Three.js Materials*
- *Chapter 5, Learning to Work with Geometries*
- *Chapter 6, Exploring Advanced Geometries*
- *Chapter 7, Points and Sprites*

4

Working with Three.js Materials

In *Chapter 3*, *Working with Light Sources in Three.js*, we talked a bit about materials. You learned that a material, together with a `THREE.Geometry` instance, forms a `THREE.Mesh` object. A material is like the skin of an object that defines what the outside of a geometry looks like. For example, a skin defines whether a geometry is metallic-looking, transparent, or shown as a wireframe. The resulting `THREE.Mesh` object can then be added to the scene to be rendered by Three.js.

So far, we haven't looked at materials in much detail. In this chapter, we'll dive into all the materials Three.js has to offer, and you'll learn how you can use these materials to create good-looking 3D objects. The materials we'll explore in this chapter are shown in the following list:

- `MeshBasicMaterial`: This is a basic material that you can use to give your geometries a simple color or show the wireframe of your geometries. This material isn't influenced by lights.
- `MeshDepthMaterial`: This is a material that uses the distance from the camera to determine how to color your mesh.
- `MeshNormalMaterial`: This is a simple material that bases the color of a face on its normal vector.
- `MeshLambertMaterial`: This is a material that takes lighting into account and is used to create dull, non-shiny-looking objects.
- `MeshPhongMaterial`: This is a material that also takes lighting into account and can be used to create shiny objects.
- `MeshStandardMaterial`: This is a material that uses physical-based rendering to render the object. With physical-based rendering, a physically correct model is used to determine how light interacts with a surface. This allows you to create more accurate and realistic-looking objects.
- `MeshPhysicalMaterial`: This is an extension of `MeshStandardMaterial` that allows more control over the reflection.
- `MeshToonMaterial`: This is an extension of `MeshPhongMaterial` that tries to make objects look hand-drawn.

- `ShadowMaterial`: This is a specific material that can receive shadows, but otherwise, it is rendered transparent.
- `ShaderMaterial`: This material allows you to specify shader programs to directly control how vertices are positioned and pixels are colored.
- `LineBasicMaterial`: This is a material that can be used on the `THREE.Line` geometry to create colored lines.
- `LineDashMaterial`: This is the same as `LineBasicMaterial`, but this material also allows you to create a dashed effect.

In the sources of Three.js, you can also find `THREE.SpriteMaterial` and `THREE.PointsMaterial`. These are materials you can use when styling individual points. We won't discuss those in this chapter, but we'll explore them in *Chapter 7*, *Points and Sprites*.

Materials have several common properties, so before we look at the first material, `THREE.MeshBasicMaterial`, we'll look at the properties shared by all the materials.

Understanding common material properties

You can quickly see for yourself which properties are shared between all the materials. Three.js provides a material base class, `THREE.Material`, that lists all these common properties. We've divided these common material properties into the following three categories:

- **Basic properties**: These are the properties you'll use most often. With these properties, you can, for instance, control the opacity of the object, whether it is visible, and how it is referenced (by ID or custom name).
- **Blending properties**: Every object has a set of blending properties. These properties define how the color of each point of the material is combined with the color behind it.
- **Advanced properties**: Several advanced properties control how the low-level WebGL context renders objects. In most cases, you won't need to deal with these properties.

Note that, in this chapter, we will skip most of the properties related to textures and maps. Most materials allow you to use images as textures (for instance, a wood-like or stone-like texture). In *Chapter 10*, *Loading and Working with Textures*, we will dive into the various available texture and mapping options. Some materials also have specific properties related to animation (for example, `skinning`, `morpNormals`, and `morphTargets`); we'll also skip those properties. These will be addressed in *Chapter 9*, *Animations and Moving the Camera*. The `clipIntersection`, `clippingPlanes`, and `clipShadows` properties will be addressed in *Chapter 6*, *Exploring Advanced Geometries*.

We will start with the first set shown in the list: the basic properties.

Basic properties

The basic properties of the `THREE.Material` object are listed here (you will see these properties in action in the *THREE.MeshBasicMaterial* section):

- `id`: This is used to identify a material and is assigned when you create a material. This starts at `0` for the first material and is increased by `1` for each additional material that is created.
- `uuid`: This is a uniquely generated ID and is used internally.
- `name`: You can assign a name to a material with this property. This can be used for debugging purposes.
- `opacity`: This defines how transparent an object is. Use this together with the `transparent` property. The range of this property is from `0` to `1`.
- `transparent`: If this is set to `true`, Three.js will render this object with the set opacity. If this is set to `false`, the object won't be transparent, just more lightly colored. This property should also be set to `true` if you use a texture that uses an alpha (transparency) channel.
- `visible`: This defines whether this material is visible. If you set this to `false`, you won't see the object in the scene.
- `side`: With this property, you can define to which side of the geometry a material is applied. The default is `THREE.Frontside`, which applies the material to the front (outside) of an object. You can also set this to `THREE.BackSide`, which applies it to the back (inside), or `THREE.DoubleSide`, which applies it to both sides.
- `needsUpdate`: When Three.js creates a material, it converts it into a set of WebGL instructions. When you want the changes you made in the material to also result in an update to the WebGL instructions, you can set this property to `true`.
- `colorWrite`: If set to `false`, the color of this material won't be shown (in effect, you'll create invisible objects, which occlude objects behind them).
- `flatShading`: This determines whether this material is rendered using flat shading. With flat shading, the individual triangles that make up an object are rendered separately and aren't combined into a smooth surface.
- `lights`: This is a Boolean value that determines whether this material is affected by lights. The default value is `true`.
- `premultipliedAlpha`: This changes the way the transparency of an object is rendered. The default value is `false`.
- `dithering`: This applies a dithering effect to the rendering material. This can be used to avoid banding. The default value is `false`.
- `shadowSide`: This is just like the `side` property but determines which side of the faces casts the shadows. If not set, this follows the value set on the `side` property.

- `vertexColors`: With this property, you can define individual colors to be applied to each vertex. If set to `true`, any color set on a vertex is used in rendering, while if set to `false`, the colors of the vertices aren't used.
- `fog`: This property determines whether this material is affected by global fog settings. This is not shown in action, but if this is set to `false`, the global fog we saw in *Chapter 2*, *Basic Components that Make up a Three.js Scene*, is disabled.

For each material, you can also set several blending properties.

Blending properties

Materials have a couple of generic blending-related properties. Blending determines how the colors we render interact with the colors that are behind them. We'll touch upon this subject a little bit when we talk about combining materials. The blending properties are listed here:

- `blending`: This determines how the material on this object blends with the background. The normal mode is `THREE.NormalBlending`, which only shows the top layer.
- `blendSrc`: Besides using the standard blending modes, you can also create custom blend modes by setting `blendsrc`, `blenddst`, and `blendequation`. This property defines how an object (the source) is blended into the background (the destination). The default `THREE.SrcAlphaFactor` setting uses the alpha (transparency) channel for blending.
- `blendSrcAlpha`: This is the transparency of `blendSrc`. The default is `null`.
- `blendDst`: This property defines how the background (the destination) is used in blending and defaults to `THREE.OneMinusSrcAlphaFactor`, which means this property also uses the alpha channel of the source for blending but uses `1` (the alpha channel of the source) as the value.
- `blendDstAlpha`: This is the transparency of `blendDst`. The default is `null`.
- `blendEquation`: This defines how the `blendsrc` and `blenddst` values are used. The default is to add them (`AddEquation`). With these three properties, you can create your own custom blend modes.

The last set of properties is mostly used internally and controls the specifics of how WebGL is used to render the scene.

Advanced properties

We won't go into the details of these properties. These are related to how WebGL works internally. If you do want to know more about these properties, the OpenGL specification is a good starting point. You can find this specification at `https://www.khronos.org/opengl/wiki`. The following list provides a brief description of these advanced properties:

- `depthTest`: This is an advanced WebGL property. With this property, you can enable or disable the `GL_DEPTH_TEST` parameter. This parameter controls whether the depth of a pixel is used to determine a new pixel's value. Normally, you wouldn't need to change this. More information can be found in the OpenGL specification we mentioned earlier.
- `depthWrite`: This is another internal property. This property can be used to determine whether this material affects the WebGL depth buffer. If you use an object for a 2D overlay (for example, a hub), you should set this property to `false`. Usually, though, you shouldn't need to change this property.
- `depthFunc`: This function compares a pixel's depth. This corresponds to `glDepthFunc` from the WebGL specifications.
- `polygonOffset`, `polygonOffsetFactor`, and `polygonOffsetUnits`: With these properties, you can control the `POLYGON_OFFSET_FILL` WebGL feature. These are normally not needed. For an explanation of what they do in detail, you can look at the OpenGL specification.
- `Alphatest`: This value can be set to a specific value (`0` to `1`). Whenever a pixel has an alpha value smaller than this value, it won't be drawn. You can use this property to remove some transparency-related artifacts. You can set the precision for this material to one of the following WebGL values: `highp`, `mediump`, or `lowp`.

Now, let's look at all the available materials so that you can see the effect these properties have on the rendered output.

Starting with simple materials

In this section, we'll look at a few simple materials: `MeshBasicMaterial`, `MeshDepthMaterial`, and `MeshNormalMaterial`.

Before we look into the properties of these materials, here's a quick note on how you can pass in properties to configure the materials. There are two options:

- You can pass in the arguments in the constructor as a parameter object, like this:

```
const material = new THREE.MeshBasicMaterial({
  color: 0xff0000,
  name: 'material-1',
  opacity: 0.5,
  transparency: true,
  ...
})
```

- Alternatively, you can create an instance and set the properties individually, like this:

  ```
  const material = new THREE.MeshBasicMaterial();
  material.color = new THREE.Color(0xff0000);
  material.name = 'material-1'; material.opacity = 0.5;
  material.transparency = true;
  ```

Usually, the best way is to use the constructor if we know all the properties' values while creating the material. The arguments used in both of these styles use the same format. The only exception to this rule is the `color` property. In the first style, we can just pass in the hex value, and Three.js will create a `THREE.Color` object itself. In the second style, we have to explicitly create a `THREE.Color` object. In this book, we'll use both of these styles.

Now, let's look at the first of the simple materials: `THREE.MeshBasicMaterial`.

THREE.MeshBasicMaterial

`MeshBasicMaterial` is a very simple material that doesn't take into account the lights that are available in the scene. Meshes with this material will be rendered as simple, flat polygons, and you also have the option to show the geometry's wireframe. Besides the common properties we saw earlier regarding this material, we can set the following properties (once again, we will ignore the properties that are used for textures since we'll discuss those in the chapter on textures):

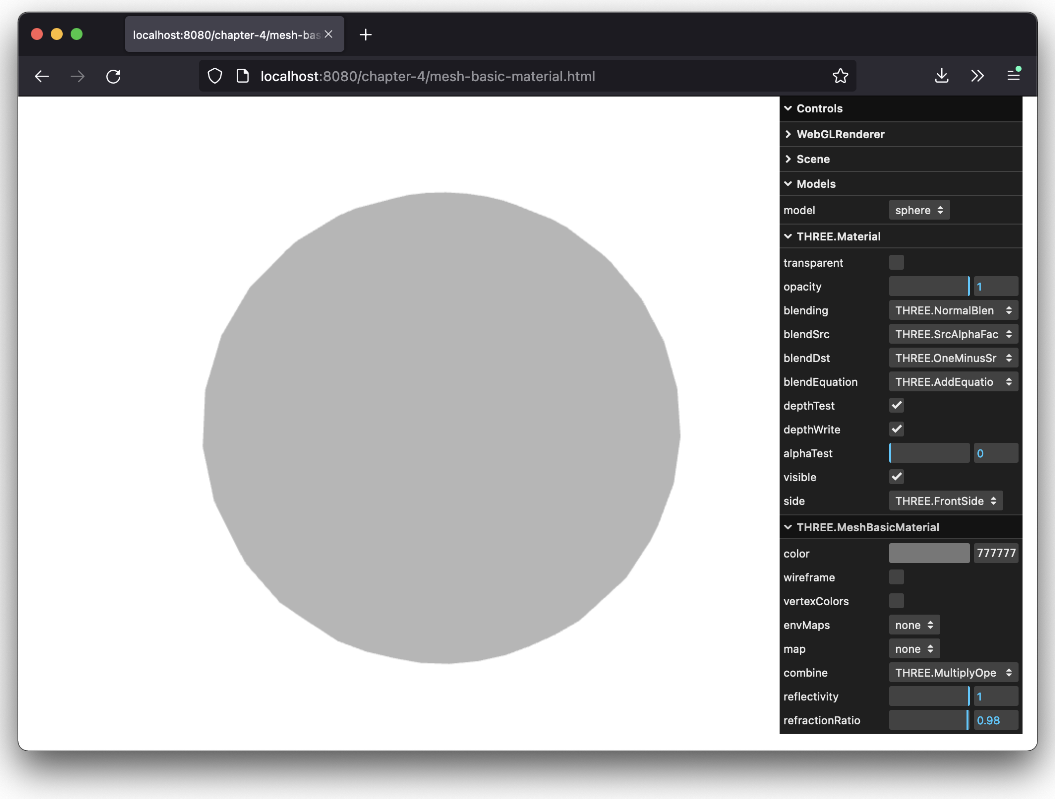
- `color`: This property allows you to set the color of the material.
- `wireframe`: This allows you to render the material as a wireframe. This is great for debugging purposes.
- `vertexColors`: When set to `true`, this will take the color of the individual vertices into account when rendering the model.

In the previous chapters, we saw how to create materials and assign them to objects. For `THREE.MeshBasicMaterial`, we can do so like this:

```
const meshMaterial = new THREE.MeshBasicMaterial({color:
  0x7777ff});
```

This creates a new `THREE.MeshBasicMaterial` and initializes the `color` property to `0x7777ff` (which is purple).

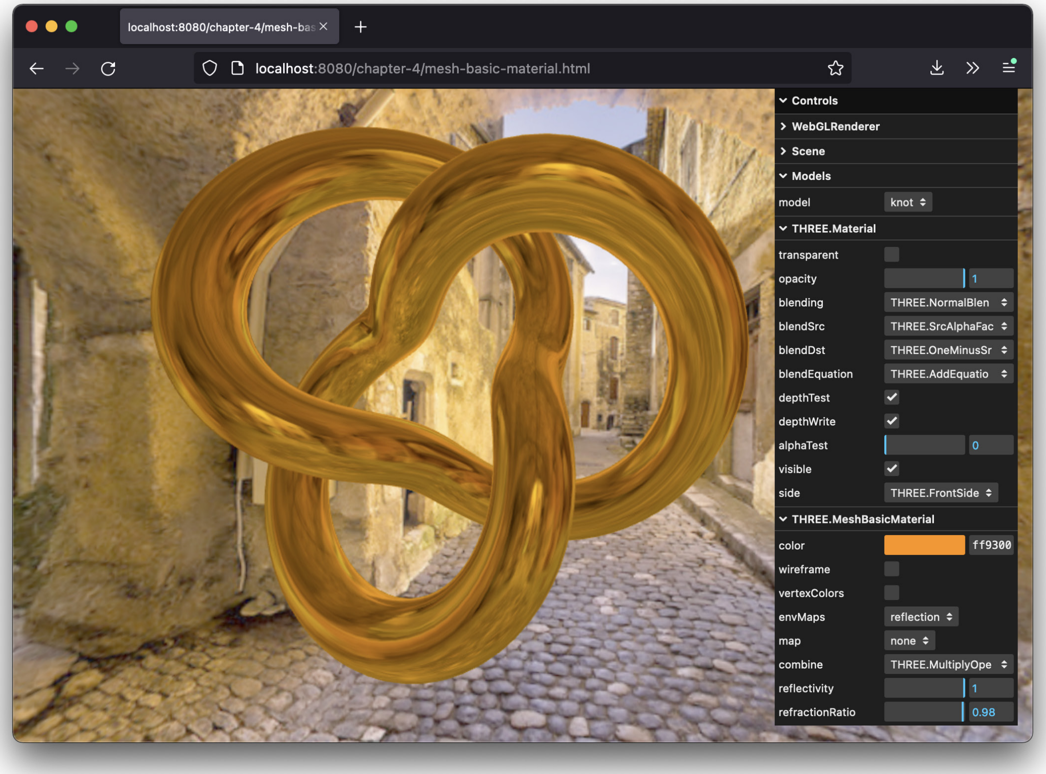
We've added an example that you can use to play around with the `THREE.MeshBasicMaterial` properties and the basic properties we discussed in the previous sections. If you open up the `basic-mesh-material.html` example in the `chapter-04` folder, you'll see a simple mesh on screen and a set of properties on the right of the scene that you can use to change models, add a simple texture, and change any of the material properties to see the effect immediately:


localhost:8080/chapter-4/mesh-bas
localhost:8080/chapter-4/mesh-basic-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshBasicMaterial

model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color 777777
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98

Figure 4.1 – Start screen for the basic material example

What you can see in this screenshot is a basic simple gray sphere. We already mentioned that `THREE.MeshBasicMaterial` doesn't respond to lights, so you don't see any depth; all the faces are the same color. Even with this material, though, you can still create nice-looking models. If you, for instance, enable the reflection by selecting the `reflection` property in the `envMaps` dropdown, set the background of the scene, and change the model to the `torus` model, you can already create great-looking models:



localhost:8080/chapter-4/mesh-basic-material.html

Controls

WebGLRenderer

Scene

Models

model	knot

THREE.Material

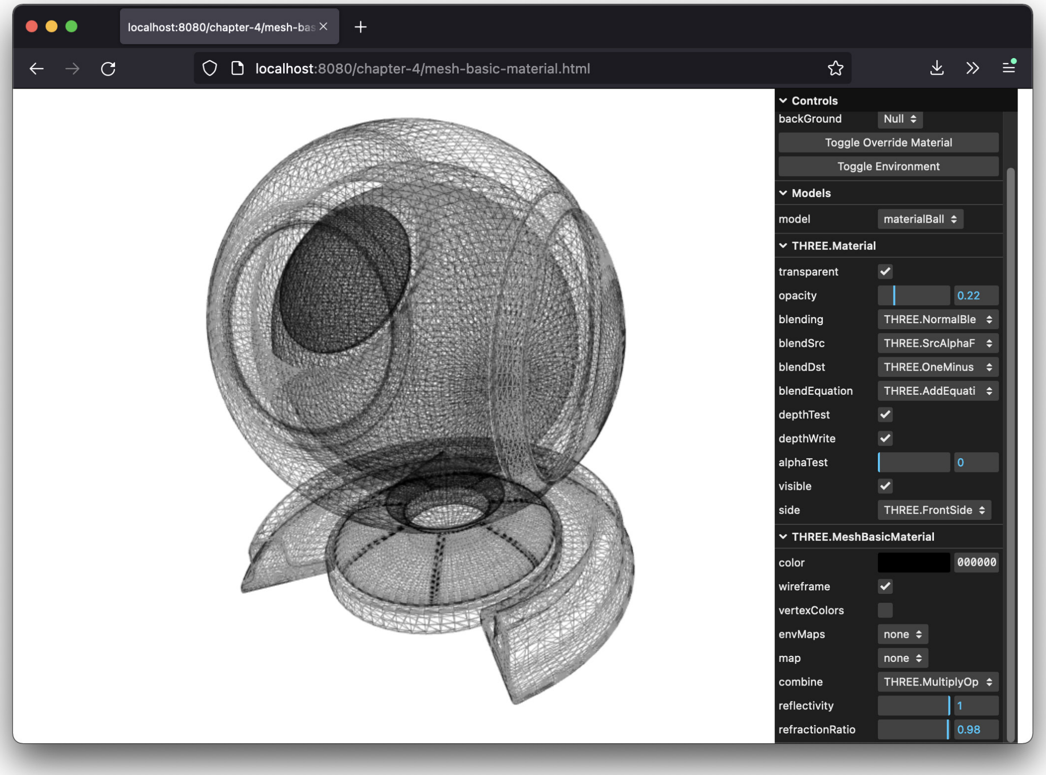
transparent	☐
opacity	1
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide

THREE.MeshBasicMaterial

color	ff9300
wireframe	☐
vertexColors	☐
envMaps	reflection
map	none
combine	THREE.MultiplyOpe
reflectivity	1
refractionRatio	0.98

Figure 4.2 – Torus knot with an environment map

The `wireframe` property is a great one for looking at the underlying geometry of `THREE.Mesh` and works great for debugging:



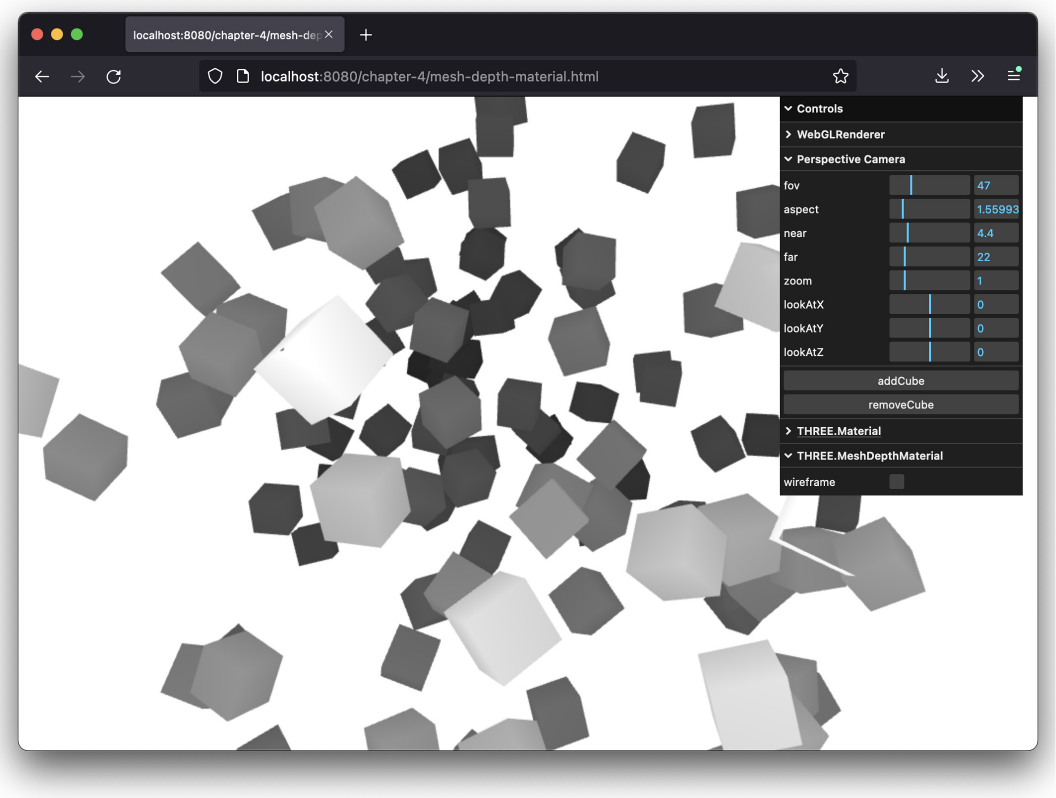
localhost:8080/chapter-4/mesh-bas
Controls
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshBasicMaterial

localhost:8080/chapter-4/mesh-basic-material.html

backGround: Null
model: materialBall
transparent: ☑
opacity: 0.22
blending: THREE.NormalBle
blendSrc: THREE.SrcAlphaF
blendDst: THREE.OneMinus
blendEquation: THREE.AddEquati
depthTest: ☑
depthWrite: ☑
alphaTest: 0
visible: ☑
side: THREE.FrontSide
color: 000000
wireframe: ☑
vertexColors: ☐
envMaps: none
map: none
combine: THREE.MultiplyOp
reflectivity: 1
refractionRatio: 0.98

Figure 4.3 – Model showing its wireframe

The final property that we want to look a bit closer at is `vertexColors`. If you enable this property, the colors of the individual vertices are used in rendering the model. If you select `vertexColor` from the model dropdown in the menu, you'll see a model that has colored vertices. The easiest way to see this is by also enabling the wireframe:


localhost:8080/chapter-4/mesh-de
localhost:8080/chapter-4/mesh-depth-material.html
Controls
WebGLRenderer
Perspective Camera
addCube
removeCube
THREE.Material
THREE.MeshDepthMaterial

fov 47
aspect 1.55993
near 4.4
far 22
zoom 1
lookAtX 0
lookAtY 0
lookAtZ 0
wireframe ☐

Figure 4.4 – Model showing wireframe and vertex colors

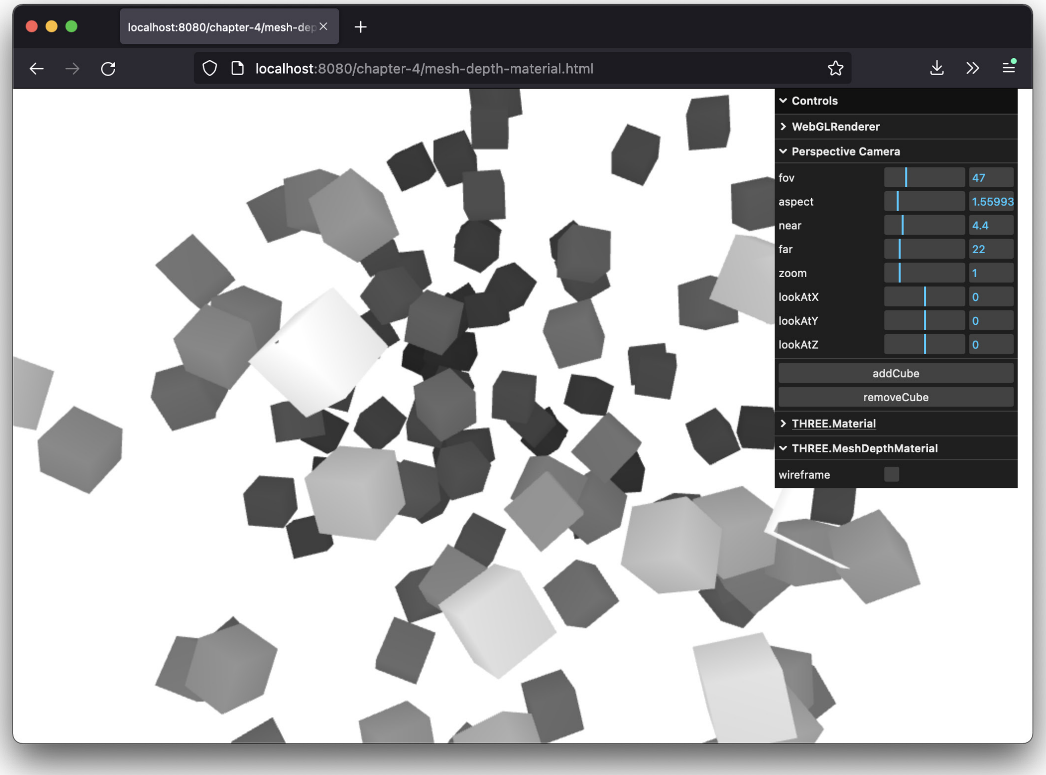
Vertex colors can be used to color different parts of the mesh in different colors without having to use textures or multiple materials.

In this example, you can also play around with the standard material properties we discussed at the beginning of this chapter by looking at the **THREE.Material** section of the menu in *Figure 4.4*.

THREE.MeshDepthMaterial

The next material on the list is `THREE.MeshDepthMaterial`. With this material, the way an object looks isn't defined by lights or by a specific material property – it is defined by the distance from the object to the camera. You can, for instance, combine this with other materials to easily create fading effects. The only additional property this material has is one we saw in `THREE.MeshBasicMaterial`: the `wireframe` property.

To demonstrate this material, we created an example that you can view by opening the `mesh-depth-material` example:


localhost:8080/chapter-4/mesh-de
localhost:8080/chapter-4/mesh-depth-material.html
Controls
WebGLRenderer
Perspective Camera
addCube
removeCube
THREE.Material
THREE.MeshDepthMaterial

fov 47
aspect 1.55993
near 4.4
far 22
zoom 1
lookAtX 0
lookAtY 0
lookAtZ 0
wireframe ☐

Figure 4.5 – Mesh depth material

In this example, you can add and remove cubes by clicking on the relevant buttons in the menu. What you'll see is that the cubes that are close to the camera are rendered very bright, and those farther away from the camera are rendered less bright. In this example, you can see how this works by playing around with the properties of the `Perspective Camera` settings. By playing around with the `far` and `near` properties of the camera, you can change the brightness of all the cubes in the scene.

Normally, you wouldn't use this material as the only material for a mesh; instead, you'd combine it with a different material. We'll see how that works in the next section.

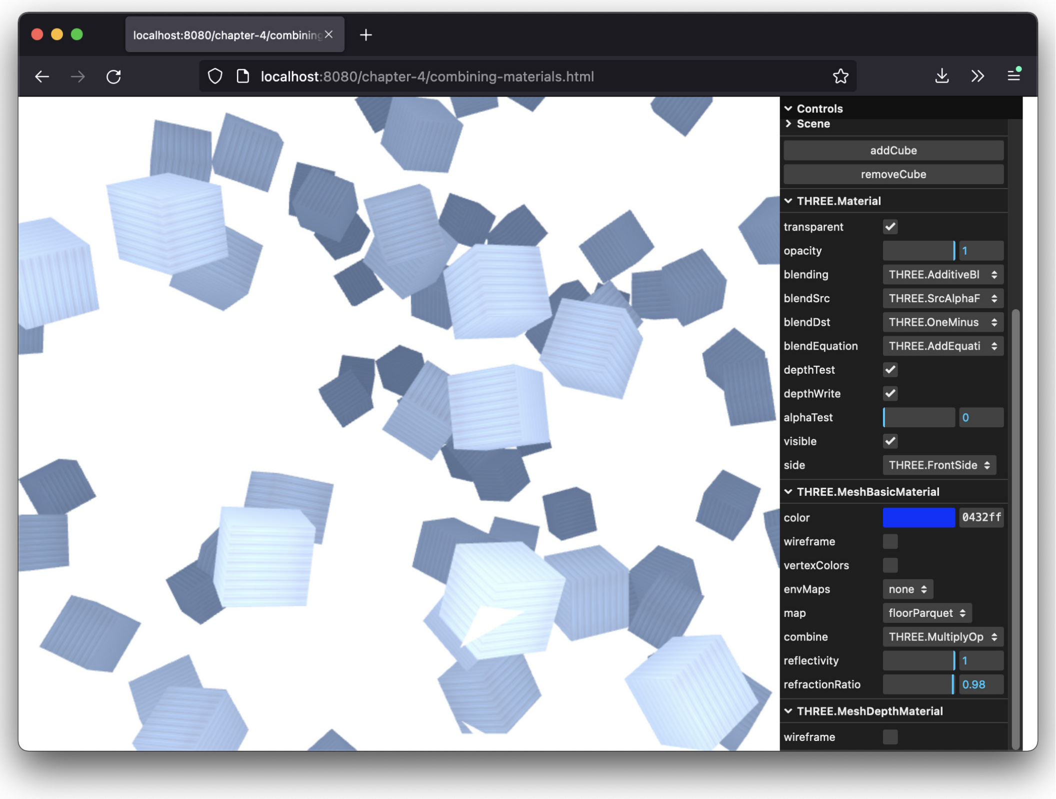
Combining materials

If you look back at the properties of `THREE.MeshDepthMaterial`, you will see that there isn't an option to set the color of the cubes. Everything was decided for you by the default properties of the material. Three.js, however, has the option to combine materials to create new effects (this is also where blending comes into play). The following code shows how we can combine materials:

```
import * as SceneUtils from 'three/examples/jsm/
  utils/SceneUtils'

const material1 = new THREE.MeshDepthMaterial()
const material2 = new THREE.MeshBasicMaterial({ color:
  0xffff00 })
const geometry = new THREE.BoxGeometry(0.5, 0.5, 0.5)
const cube = SceneUtils.createMultiMaterialObject(geometry,
  [material2, material1])
```

First, we create our two materials. For `THREE.MeshDepthMaterial`, we don't do anything special; for `THREE.MeshBasicMaterial`, we just set the color. The last line in this code fragment is also an important one. When we create a mesh with the `SceneUtils.createMultiMaterialObject()` function, the geometry gets copied and two of the same meshes are returned in a group.

We get the following green-colored cubes that use the brightness from `THREE.MeshDepthMaterial` and the color from `THREE.MeshBasicMaterial`. You can see how this works by opening the `combining-materials.html` example in the `chapter-4` folder in your browser:


localhost:8080/chapter-4/combining
localhost:8080/chapter-4/combining-materials.html
Controls
Scene
addCube
removeCube
THREE.Material
THREE.MeshBasicMaterial
THREE.MeshDepthMaterial

transparent ☑
opacity 1
blending THREE.AdditiveBl
blendSrc THREE.SrcAlphaF
blendDst THREE.OneMinus
blendEquation THREE.AddEquati
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color 0432ff
wireframe ☐
vertexColors ☐
envMaps none
map floorParquet
combine THREE.MultiplyOp
reflectivity 1
refractionRatio 0.98
wireframe ☐

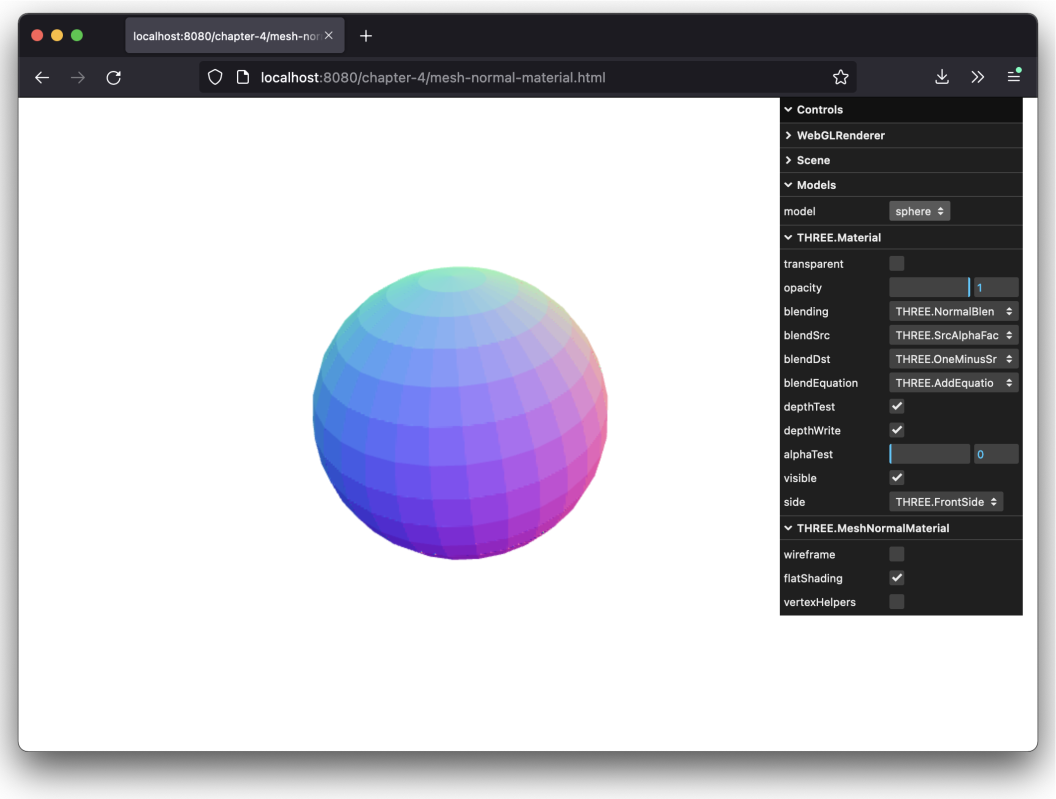
Figure 4.6 – Combining materials

When you open this example for the first time, you'll just see the solid objects, without any effect from `THREE.MeshDepthMaterial`. To combine the colors, we also need to specify how these colors blend. In the menu on the right in *Figure 4.6*, you can specify this using the `blending` property. For this example, we've used the `THREE.AdditiveBlending` mode, which means the colors are added together, and the resulting color is shown. This example is a great way to play around with the different blending options, and see how they affect the final color of the material.

The next material is also one where we won't have any influence on the colors used in rendering.

THREE.MeshNormalMaterial

The easiest way to understand how this material is rendered is by first looking at an example. Open up the `mesh-normal-material.html` example in the `chapter-4` folder and enable `flatShading`:

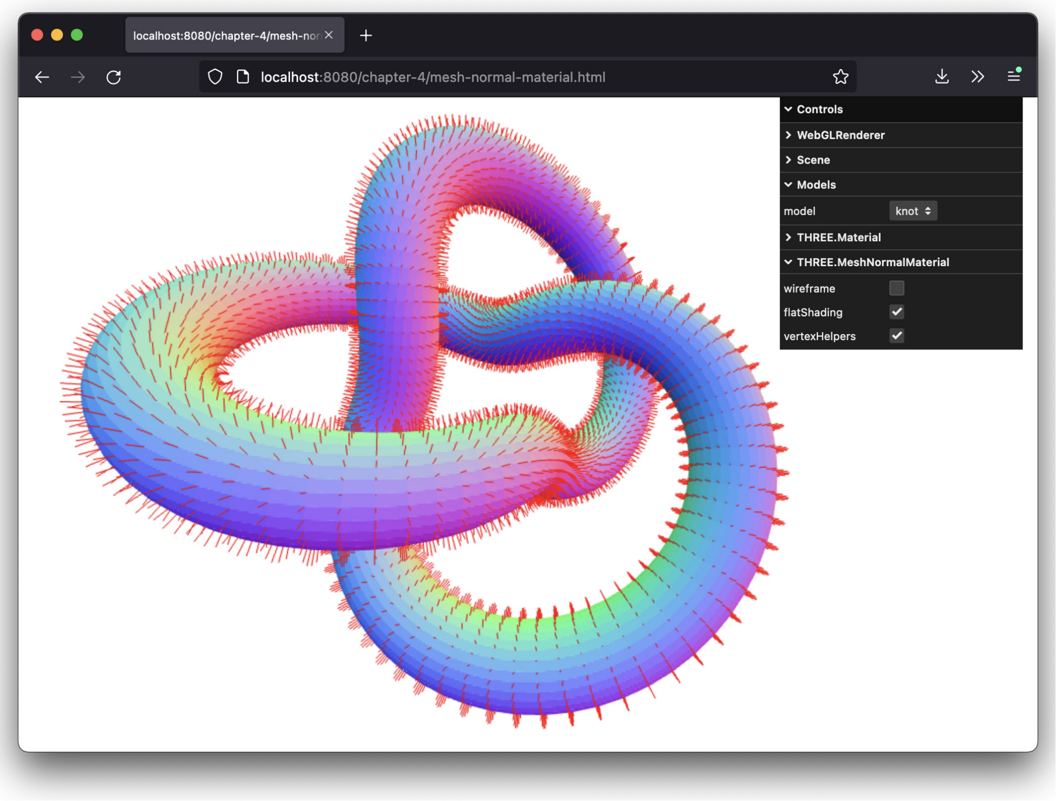

localhost:8080/chapter-4/mesh-nor
localhost:8080/chapter-4/mesh-normal-material.html

Controls
WebGLRenderer
Scene
Models
model sphere
THREE.Material
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
THREE.MeshNormalMaterial
wireframe ☐
flatShading ☑
vertexHelpers ☐

Figure 4.7 – Mesh normal material

As you can see, each face of the mesh is rendered in a slightly different color. This happens because the color of each face is based on the normals pointing out from the face. And this face normal is based on the normal vector of the individual vertices that make up the face. A normal vector is perpendicular to the face of a vertex. The normal vector is used in many different parts of Three.js. It is used to determine light reflections, helps with mapping textures to 3D models, and provides information on how to light, shade, and color pixels on the surface. Luckily, though, Three.js handles the computation of these vectors and uses them internally, so you don't have to calculate or deal with them yourselves.

Three.js comes with a helper to visualize this normal, and you can show this by enabling the `vertexHelpers` property in the menu:


localhost:8080/chapter-4/mesh-nor
localhost:8080/chapter-4/mesh-normal-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshNormalMaterial

model knot

wireframe ☐

flatShading ☑

vertexHelpers ☑

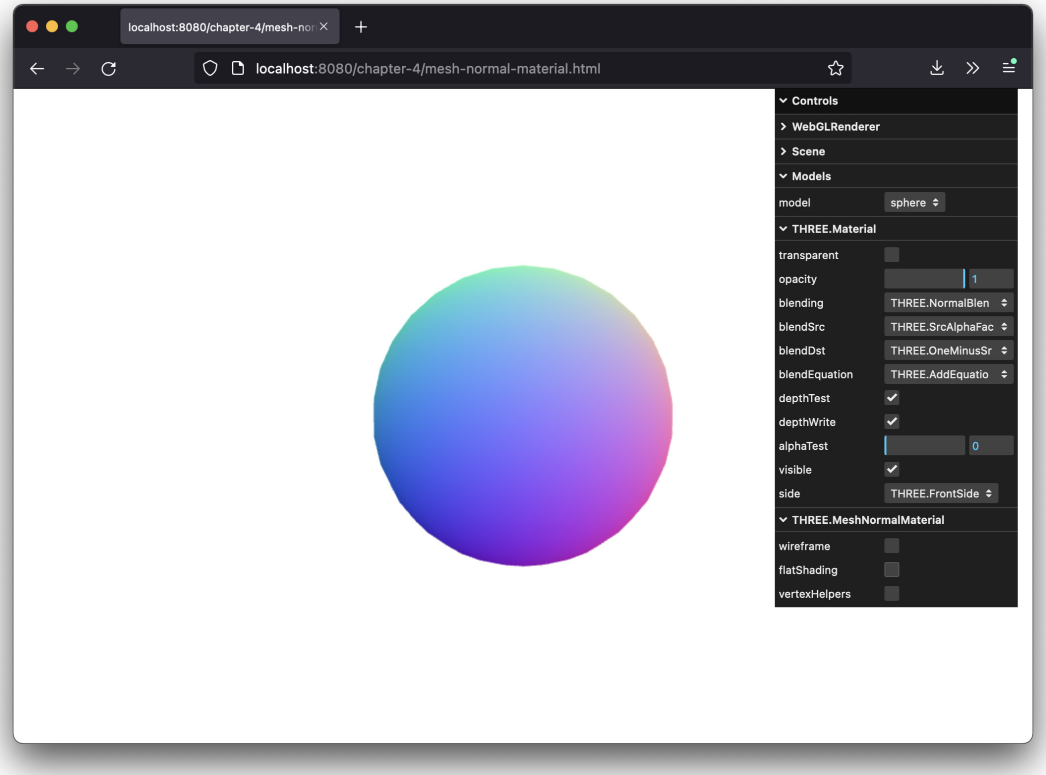
Figure 4.8 – Mesh normal helpers

Adding this helper yourself can be done in a couple of lines of code:

```
Import { VertexNormalsHelper } from 'three/examples/jsm/
  helpers/VertexNormalsHelper'
...
const helper = new VertexNormalsHelper(mesh, 0.1, 0xff0000)
helper.name = 'VertexNormalHelper'
scene.add(helper)
```

`VertexNormalsHelper` takes three parameters. The first one is `THREE.Mesh`, for which you want to see the helpers, the second one is the length of the arrow, and the last one is the color.

Let's take this example as an opportunity to look at the `shading` property. With the `shading` property, we can tell Three.js how to render our objects. If you use `THREE.FlatShading`, each face will be rendered as-is (as you can see in the previous following screenshot), or you can use `THREE.SmoothShading`, which smooths out the faces of our objects. For instance, if we render the same sphere using `THREE.SmoothShading`, the result will look like this:


localhost:8080/chapter-4/mesh-nor
localhost:8080/chapter-4/mesh-normal-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshNormalMaterial

model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
wireframe ☐
flatShading ☐
vertexHelpers ☐

Figure 4.9 – Mesh normal smooth shading

We're done with the simple materials, but let's look at one additional subject before moving on. In the next section, we'll look at how you can use different materials for specific faces of a geometry.

Multiple materials for a single mesh

When creating `THREE.Mesh`, so far, we've used a single material. It is also possible to define a specific material for each of the faces of a geometry. For instance, if you have a cube that has 12 faces (remember, Three.js works with triangles), you can assign a different material (for example, with a different color) to each side of the cube. Doing this is straightforward, as shown in the following piece of code:

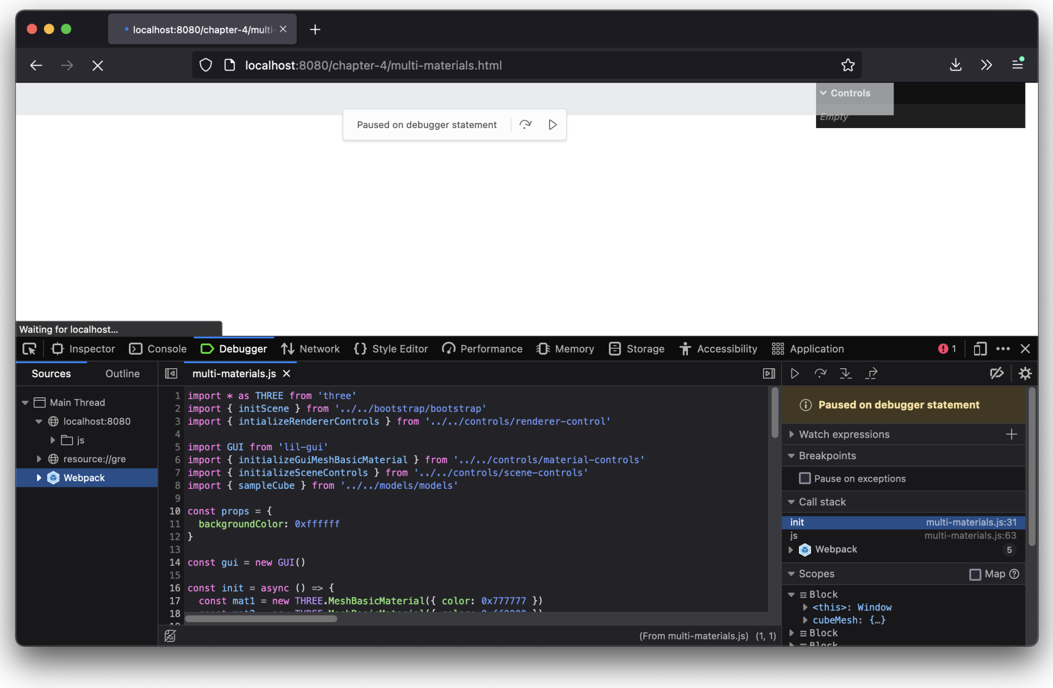
```
const mat1 = new THREE.MeshBasicMaterial({ color: 0x777777
  })
const mat2 = new THREE.MeshBasicMaterial({ color: 0xff0000
  })
const mat3 = new THREE.MeshBasicMaterial({ color: 0x00ff00
  })
const mat4 = new THREE.MeshBasicMaterial({ color: 0x0000ff
  })
const mat5 = new THREE.MeshBasicMaterial({ color: 0x66aaff
  })
const mat6 = new THREE.MeshBasicMaterial({ color: 0xffaa66
  })
const matArray = [mat1, mat2, mat3, mat4, mat5, mat6]
const cubeGeom = new THREE.BoxGeometry(1, 1, 1, 10, 10, 10)
const cubeMesh = new THREE.Mesh(cubeGeom, material)
```

We create an array, named `matArray`, to hold all the materials, and use that array to create `THREE.Mesh`. What you might notice is that we only create six materials, even though we've got 12 faces. To understand how this works, we have to look at how Three.js assigns a material to a face. Three.js uses the `groups` property for this. To see this yourself, open up the source code for `multi-material.js` and add the `debugger` statement, like this:

```
  const group = new THREE.Group()
  for (let x = 0; x < 3; x++) {
    for (let y = 0; y < 3; y++) {
      for (let z = 0; z < 3; z++) {
        const cubeMesh = sampleCube([mat1, mat2, mat3,
          mat4, mat5, mat6], 0.95)
        cubeMesh.position.set(x - 1.5, y - 1.5, z - 1.5)
        group.add(cubeMesh)
        debugger
```

```
            }
        }
    }
```

This will cause the browser to stop executing, and allows you to inspect all the objects from the console of the browser:

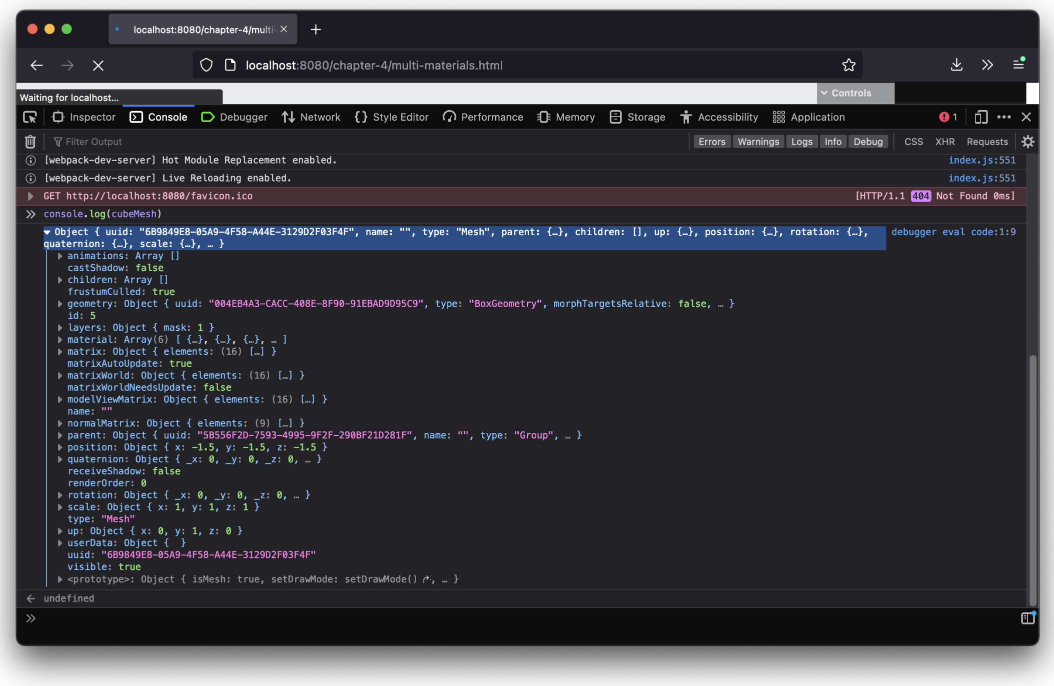


```
import * as THREE from 'three'
import { initScene } from '../../bootstrap/bootstrap'
import { intializeRendererControls } from '../../controls/renderer-control'

import GUI from 'lil-gui'
import { initializeGuiMeshBasicMaterial } from '../../controls/material-controls'
import { initializeSceneControls } from '../../controls/scene-controls'
import { sampleCube } from '../../models/models'

const props = {
  backgroundColor: 0xffffff
}

const gui = new GUI()

const init = async () => {
  const mat1 = new THREE.MeshBasicMaterial({ color: 0x777777 })
```

localhost:8080/chapter-4/multi-materials.html
Paused on debugger statement
Controls
Empty
Waiting for localhost...
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Sources
Outline
Main Thread
localhost:8080
js
resource://gre
Webpack
multi-materials.js
(From multi-materials.js) (1, 1)
Paused on debugger statement
Watch expressions
Breakpoints
Pause on exceptions
Call stack
init multi-materials.js:31
js multi-materials.js:63
Webpack 5
Scopes
Map
Block
<this>: Window
cubeMesh: {…}
Block

Figure 4.10 – Using the debugger statement to stop execution

In the browser, if you open the **Console** tab, you can print our information about all different kinds of objects. So, if we want to see the details of `cubeMesh`, we can use `console.log(cubeMesh)`:


localhost:8080/chapter-4/multi
localhost:8080/chapter-4/multi-materials.html
Waiting for localhost...
Controls
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Filter Output
Errors
Warnings
Logs
Info
Debug
CSS
XHR
Requests

```
[webpack-dev-server] Hot Module Replacement enabled.                                                      index.js:551
[webpack-dev-server] Live Reloading enabled.                                                              index.js:551
GET http://localhost:8080/favicon.ico                                                [HTTP/1.1 404 Not Found 0ms]
>> console.log(cubeMesh)
▼ Object { uuid: "6B9849E8-05A9-4F58-A44E-3129D2F03F4F", name: "", type: "Mesh", parent: {…}, children: [], up: {…}, position: {…}, rotation: {…},   debugger eval code:1:9
quaternion: {…}, scale: {…}, … }
  ▶ animations: Array []
    castShadow: false
  ▶ children: Array []
    frustumCulled: true
  ▶ geometry: Object { uuid: "004EB4A3-CACC-408E-8F90-91EBAD9D95C9", type: "BoxGeometry", morphTargetsRelative: false, … }
    id: 5
  ▶ layers: Object { mask: 1 }
  ▶ material: Array(6) [ {…}, {…}, {…}, … ]
  ▶ matrix: Object { elements: (16) […] }
    matrixAutoUpdate: true
  ▶ matrixWorld: Object { elements: (16) […] }
    matrixWorldNeedsUpdate: false
  ▶ modelViewMatrix: Object { elements: (16) […] }
    name: ""
  ▶ normalMatrix: Object { elements: (9) […] }
  ▶ parent: Object { uuid: "5B556F2D-7593-4995-9F2F-290BF21D281F", name: "", type: "Group", … }
  ▶ position: Object { x: -1.5, y: -1.5, z: -1.5 }
  ▶ quaternion: Object { _x: 0, _y: 0, _z: 0, … }
    receiveShadow: false
    renderOrder: 0
  ▶ rotation: Object { _x: 0, _y: 0, _z: 0, … }
  ▶ scale: Object { x: 1, y: 1, z: 1 }
    type: "Mesh"
  ▶ up: Object { x: 0, y: 1, z: 0 }
  ▶ userData: Object {  }
    uuid: "6B9849E8-05A9-4F58-A44E-3129D2F03F4F"
    visible: true
  ▶ <prototype>: Object { isMesh: true, setDrawMode: setDrawMode() ↱, … }
← undefined
>>
```

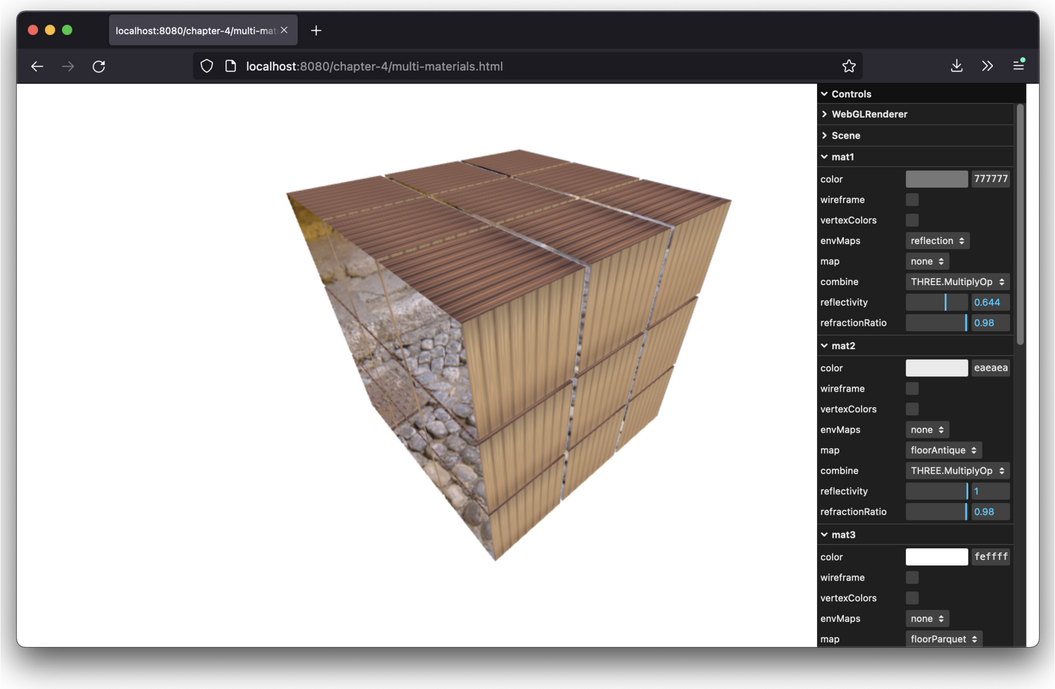
Figure 4.11 – Printing out information about an object

If you look further into the `geometry` property of `cubeMesh`, you will see `groups`. This property is an array that consists of six elements, where each element contains the range of vertices that belong to that group, and an additional property called `materialIndex` that specifies which of the passed-in materials should be used for that group of vertices:

```
[{ "start": 0,    "count": 600, "materialIndex": 0 },
 { "start": 600,  "count": 600, "materialIndex": 1 },
 { "start": 1200, "count": 600, "materialIndex": 2 },
 { "start": 1800, "count": 600, "materialIndex": 3 },
 { "start": 2400, "count": 600, "materialIndex": 4 },
 { "start": 3000, "count": 600, "materialIndex": 5 }]
```

So, if you create your own objects from scratch, and want to apply different materials to different vertices groups, you have to make sure you set the `groups` property correctly. For the objects created by Three.js, you don't have to do this manually, since Three.js already does this.

With this approach, it's very simple to create interesting models. For instance, we can easily create a simple 3D Rubik's Cube, as you can see in the `multi-materials.html` example:


localhost:8080/chapter-4/multi-mat ×
localhost:8080/chapter-4/multi-materials.html

Controls
WebGLRenderer
Scene
mat1
color 777777
wireframe ☐
vertexColors ☐
envMaps reflection
map none
combine THREE.MultiplyOp
reflectivity 0.644
refractionRatio 0.98
mat2
color eaeaea
wireframe ☐
vertexColors ☐
envMaps none
map floorAntique
combine THREE.MultiplyOp
reflectivity 1
refractionRatio 0.98
mat3
color feffff
wireframe ☐
vertexColors ☐
envMaps none
map floorParquet

Figure 4.12 – Multi-material with six different materials

We've also added controls for the materials that are applied to each side to experiment with. Creating this cube is not much different than what we saw in the *Multiple materials for a single mesh* section:

```
const group = new THREE.Group()
const mat1 = new THREE.MeshBasicMaterial({ color: 0x777777
  })
const mat2 = new THREE.MeshBasicMaterial({ color: 0xff0000
  })
const mat3 = new THREE.MeshBasicMaterial({ color: 0x00ff00
  })
const mat4 = new THREE.MeshBasicMaterial({ color: 0x0000ff
  })
const mat5 = new THREE.MeshBasicMaterial({ color: 0x66aaff
  })
```

```
const mat6 = new THREE.MeshBasicMaterial({ color: 0xffaa66
  })
for (let x = 0; x < 3; x++) {
  for (let y = 0; y < 3; y++) {
    for (let z = 0; z < 3; z++) {
      const cubeMesh = sampleCube([mat1, mat2, mat3, mat4,
        mat5, mat6], 0.95)
      cubeMesh.position.set(x - 1.5, y - 1.5, z - 1.5)
      group.add(cubeMesh)
    }
  }
}
```

In this piece of code, first, we create `THREE.Group`, which will hold all the individual cubes (group); next, we create the materials for each side of the cube. Then, we create three loops to make sure we create the right number of cubes. In this loop, we create each of the individual cubes, assign the materials, position them, and add them to the group. What you should remember is that the position of the cubes is relative to the position of this group. If we move or rotate the group, all the cubes will move and rotate with it. For more information on how to work with groups, look at *Chapter 8*, *Creating and Loading Advanced Meshes and Geometries*.

And that wraps up this section on basic materials and how to combine them. In the following section, we'll look at more advanced materials.

Advanced materials

In this section, we'll look at the more advanced materials Three.js has to offer. We'll look at the following materials:

- `THREE.MeshLambertMaterial`: A material for rough-looking surfaces
- `THREE.MeshPhongMaterial`: A material for shiny-looking surfaces
- `THREE.MeshToonMaterial`: Renders the mesh in a cartoon-like fashion
- `THREE.ShadowMaterial`: A material that only shows shadows cast on it; the material is otherwise is transparent
- `THREE.MeshStandardMaterial`: A versatile material that can be used to represent many different kinds of surfaces

- `THREE.MeshPhysicalMaterial`: Similar to `THREE.MeshStandardMaterial` but provides additional properties for more real-world-like surfaces
- `THREE.ShaderMaterial`: A material where you can define for yourself how to render the object by writing your own shaders

We'll start with `THREE.MeshLambertMaterial`.

THREE.MeshLambertMaterial

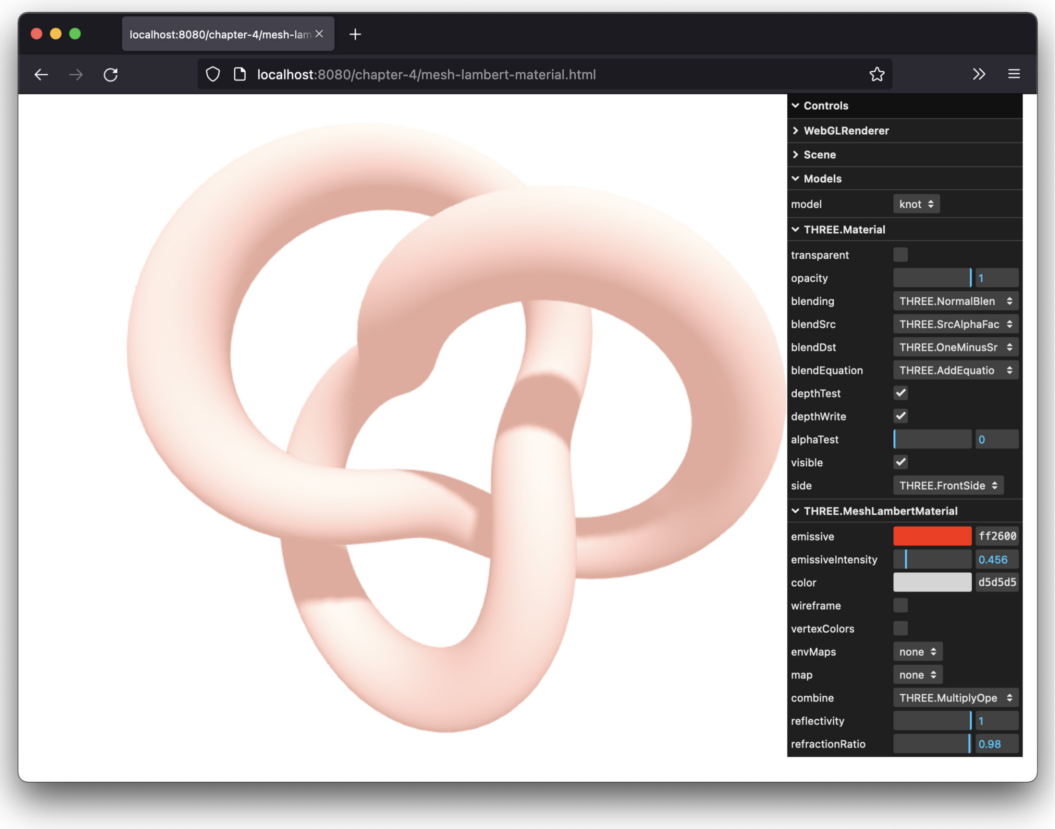
This material can be used to create dull-looking, non-shiny surfaces. This is a very easy-to-use material that responds to the lighting sources in the scene. This material can be configured with the basic properties we've already seen, so we won't go into the details of those properties; instead, we will focus on the ones specific to this material. That just leaves us with the following properties:

- `color`: This is the color of the material.
- `emissive`: This is the color the material emits. It doesn't act as a light source, but this is a solid color that is unaffected by other lighting. This defaults to black. You can use this to create objects that look like they glow.
- `emissiveIntensity`: The intensity with which the object seems to glow.

Creating this object follows the same approach we've seen for the other materials:

```
const material = new THREE.MeshLambertMaterial({color:
  0x7777ff});
```

For an example of this material, look at the `mesh-lambert-material.html` example:

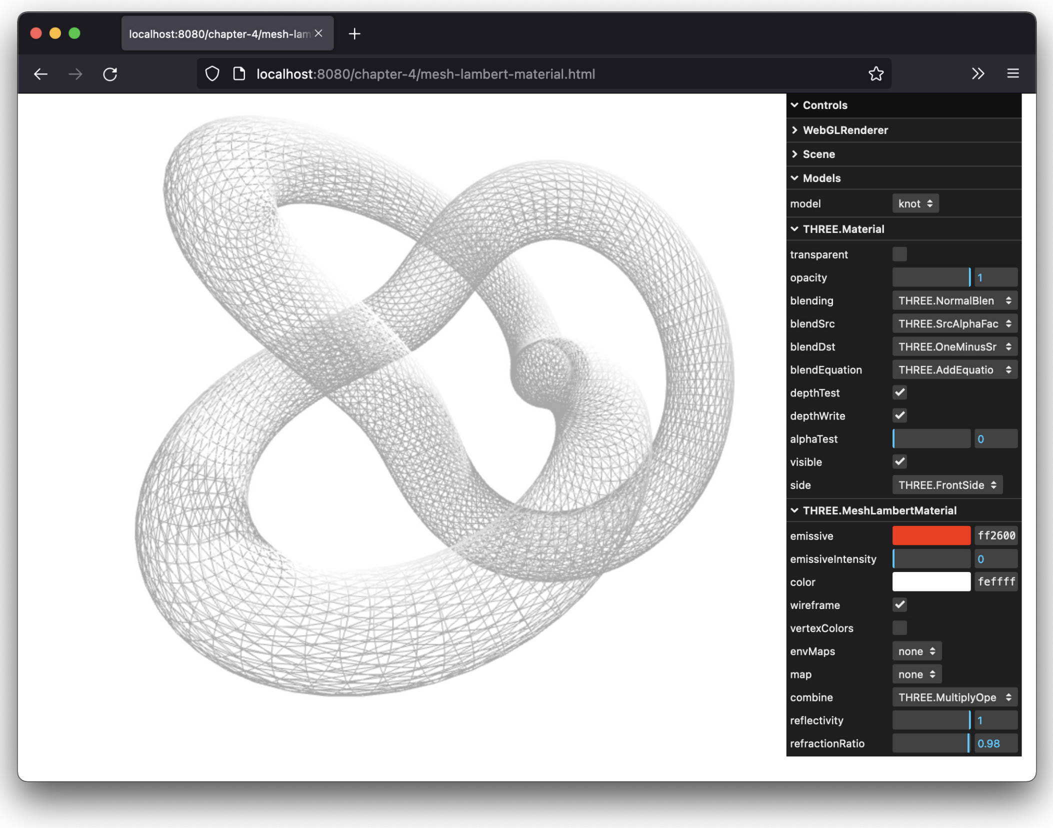


localhost:8080/chapter-4/mesh-lam
localhost:8080/chapter-4/mesh-lambert-material.html

Controls
WebGLRenderer
Scene
Models
model knot
THREE.Material
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
THREE.MeshLambertMaterial
emissive ff2600
emissiveIntensity 0.456
color d5d5d5
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98

Figure 4.13 – Mesh lambert material

This screenshot shows a torus knot, in white, with a very light red emissive glow. One of the interesting features of `THREE.LambertMaterial` is that it also supports the wireframe properties, so you can render a wireframe that responds to the lights in the scene:


localhost:8080/chapter-4/mesh-lam
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshLambertMaterial

localhost:8080/chapter-4/mesh-lambert-material.html

model knot
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive ff2600
emissiveIntensity 0
color feffff
wireframe ☑
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98

Figure 4.14 – Mesh Lambert material with a wireframe

The next material works in pretty much the same way but can be used to create shiny objects.

THREE.MeshPhongMaterial

With `THREE.MeshPhongMaterial`, we can create a shiny material. The properties you can use for that are pretty much the same as for a non-shiny `THREE.MeshLambertMaterial` object. In older versions, this was the only material that you could use to make shiny, plastic, or metal-like objects. With newer versions of Three.js, if you want more control, you can also use `THREE.MeshStandardMaterial` and `THREE.MeshPhysicalMaterial`. We'll discuss both of these materials after we look at `THREE.MeshPhongMaterial`.

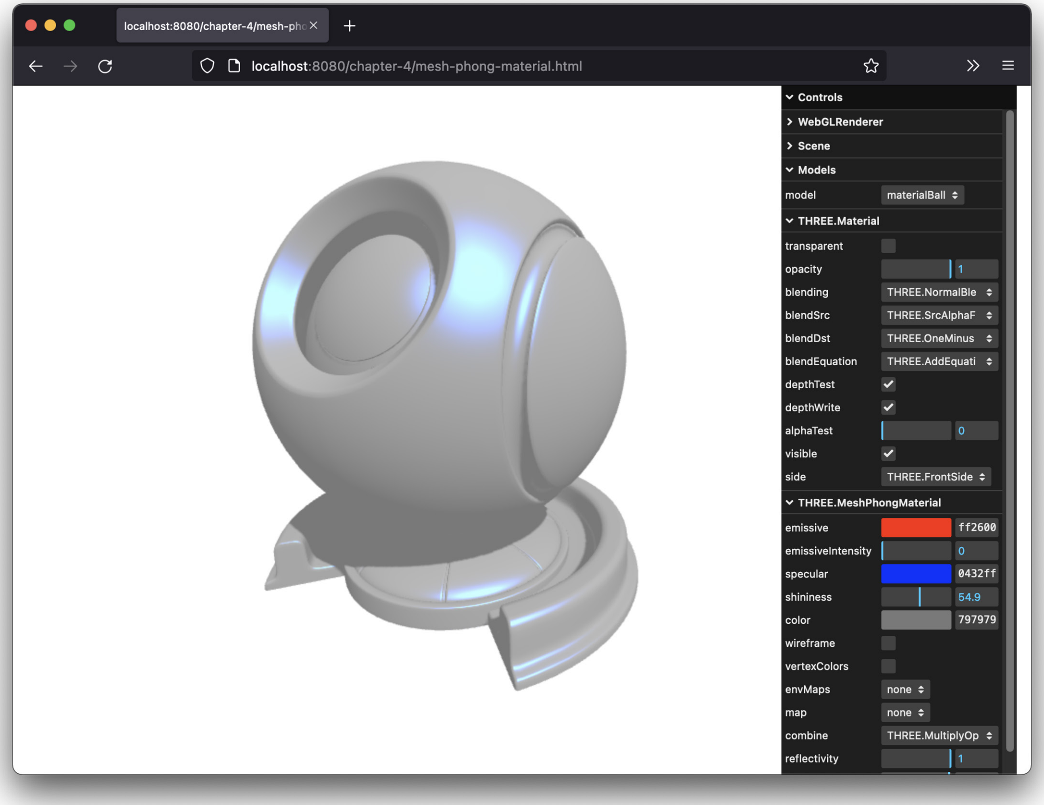
We'll once again skip the basic properties and focus on the properties specific to this material. The properties of this material are listed here:

- `emissive`: This is the color this material emits. It doesn't act as a light source, but this is a solid color that is unaffected by other lighting. This defaults to black.
- `emissiveIntensity`: The intensity with which the object seems to glow.
- `specular`: This property defines how shiny the material is and with what color it shines. If this is set to the same color as the `color` property, you get a more metallic-looking material. If this is set to gray, it results in a more plastic-looking material.
- `shininess`: This property defines how shiny the specular highlight is. The default value for `shininess` is `30`. The higher this value is, the shinier the object is.

Initializing a `THREE.MeshPhongMaterial` material is done in the same way as we've already seen for all the other materials and is shown in the following line of code:

```
const meshMaterial = new THREE.MeshPhongMaterial({color:
  0x7777ff});
```

To give you the best comparison, we will keep using the same models for this material as we did for `THREE.MeshLambertMaterial` and the other materials in this chapter. You can use the control GUI to play around with this material. For instance, the following settings create a plastic-looking material. You can find this example in `mesh-phong-material.html`:



localhost:8080/chapter-4/mesh-phong-material.html

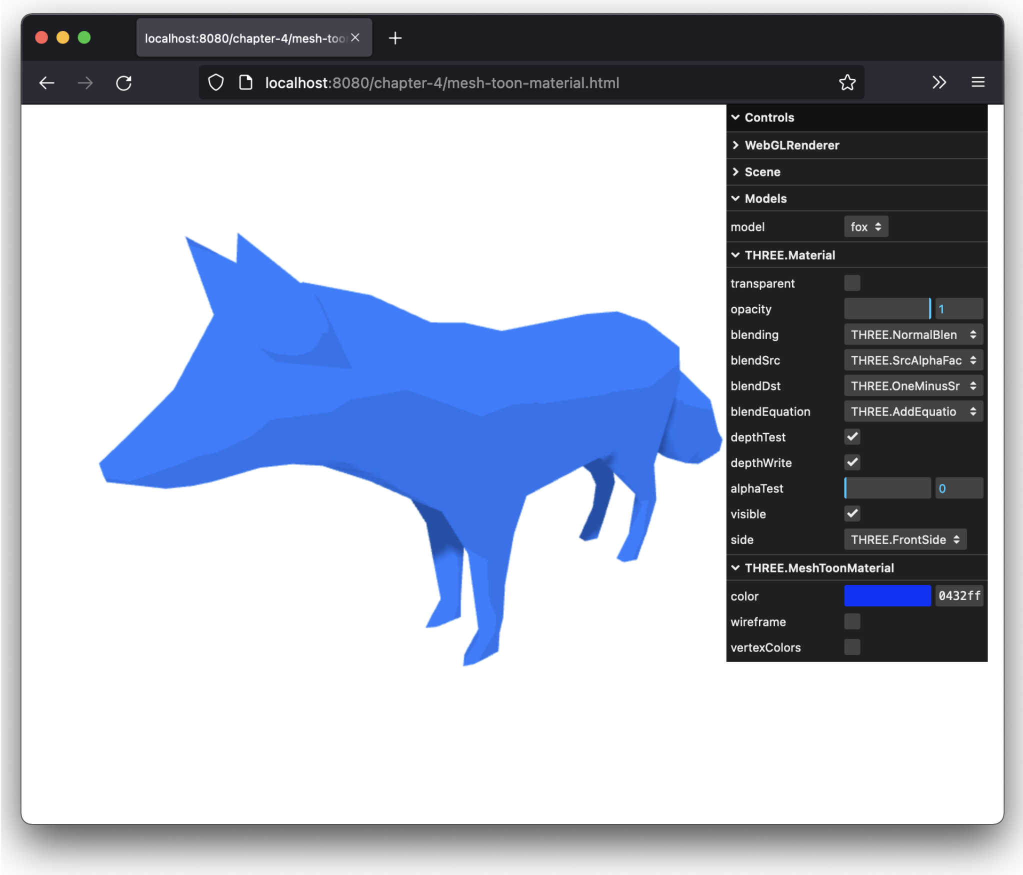
Controls
WebGLRenderer
Scene
Models
model: materialBall
THREE.Material
transparent: ☐
opacity: 1
blending: THREE.NormalBle
blendSrc: THREE.SrcAlphaF
blendDst: THREE.OneMinus
blendEquation: THREE.AddEquati
depthTest: ☑
depthWrite: ☑
alphaTest: 0
visible: ☑
side: THREE.FrontSide
THREE.MeshPhongMaterial
emissive: ff2600
emissiveIntensity: 0
specular: 0432ff
shininess: 54.9
color: 797979
wireframe: ☐
vertexColors: ☐
envMaps: none
map: none
combine: THREE.MultiplyOp
reflectivity: 1

Figure 4.15 – Mesh Phong material with high shininess

As you can see from this screenshot, the object is more shiny and plastic compared to what we saw with `THREE.MeshLambertMaterial`.

THREE.MeshToonMaterial

Not all the materials Three.js provides are practical. For instance, `THREE.MeshToonMaterial` allows you to render an object in a cartoon-like style (see the `mesh-toon-material.html` example):


localhost:8080/chapter-4/mesh-toon-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshToonMaterial

model fox
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color 0432ff
wireframe ☐
vertexColors ☐

Figure 4.16 – The fox model rendered with MeshToonMaterial

As you can see, it looks a little bit like what we saw with `THREE.MeshBasicMaterial`, but this material responds to the lights in the scene and supports shadows. It just bands colors together to create a cartoon-like effect.

If you want more realistic materials, `THREE.MeshStandardMaterial` is a good choice.

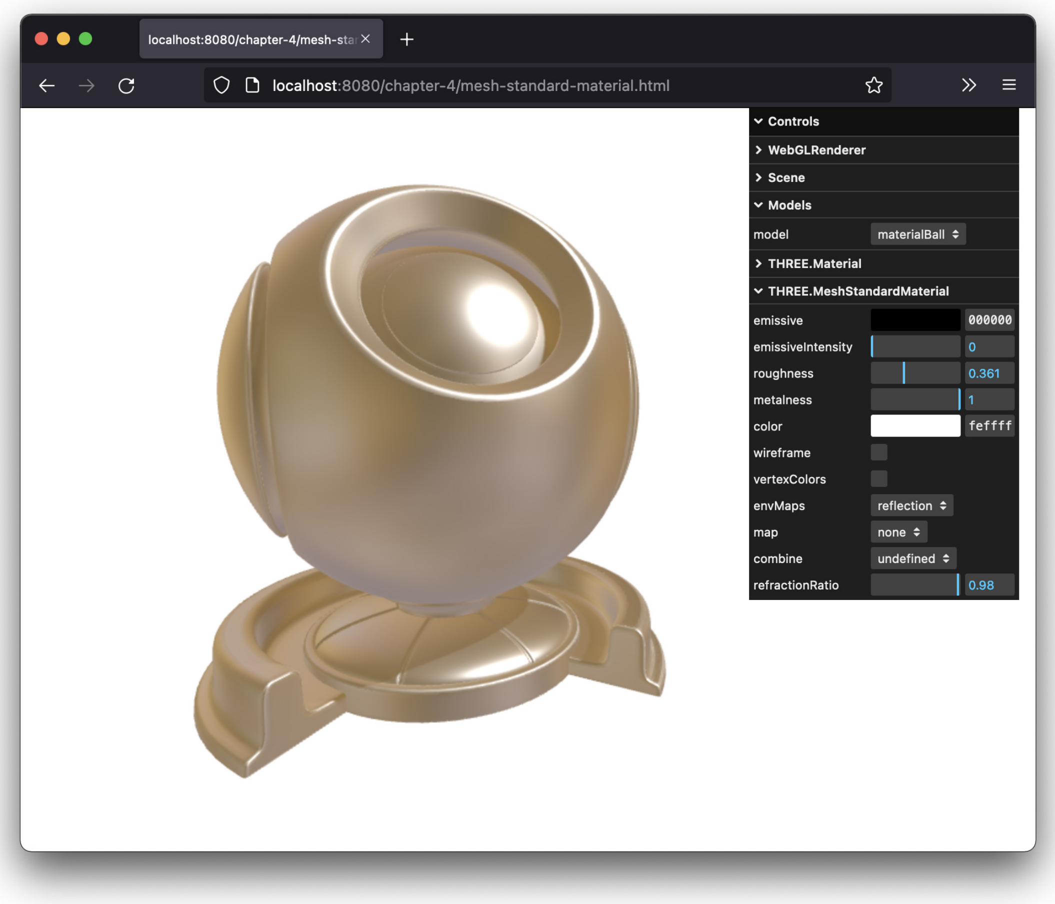
THREE.MeshStandardMaterial

`THREE.MeshStandardMaterial` is a material that takes a physics approach to determine how to react to the lighting in the scene. It is a great material for shiny and metal-like materials, and provides several properties you can use to configure this material:

- `metalness`: This property determines how metal-like a material is. Non-metallic materials should use a value of `0`, whereas metallic materials should use a value close to `1`. The default is `0.5`.

- `roughness`: You can also set how rough the material is. This determines how the light that hits this material is diffused. The default is `0.5`. A value of `0` is a mirror-like reflection, whereas a value of `1` diffuses all the light.

Besides these properties, you can also use the `color` and `emissive` properties, as well as the properties from `THREE.Material`, to alter this material. As you can see in the following screenshot, we can use these properties to simulate a kind of brushed metal look by playing around with the `metalness` and `roughness` parameters:


localhost:8080/chapter-4/mesh-sta
localhost:8080/chapter-4/mesh-standard-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshStandardMaterial

model materialBall
emissive 000000
emissiveIntensity 0
roughness 0.361
metalness 1
color feffff
wireframe ☐
vertexColors ☐
envMaps reflection
map none
combine undefined
refractionRatio 0.98

Figure 4.17 – Creating a brushed metal effect with MeshStandardMaterial

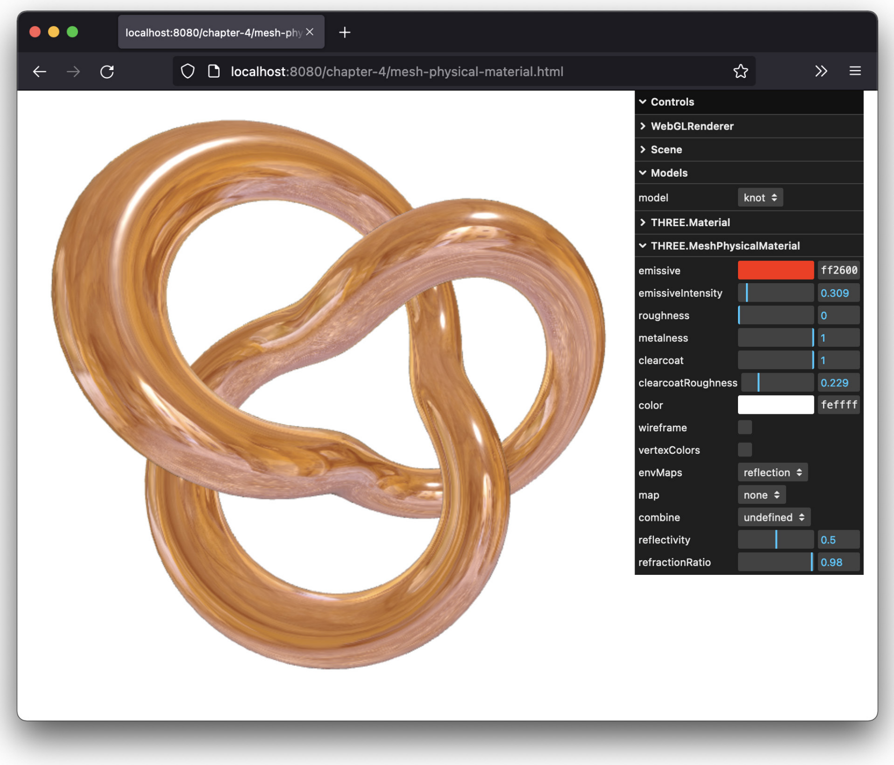
Three.js provides a material that provides even more settings to render real-looking objects: `THREE.MeshPhysicalMaterial`.

THREE.MeshPhysicalMaterial

A material very close to `THREE.MeshStandardMaterial` is `THREE.MeshPhysicalMaterial`. With this material, you have more control over the reflectivity of the material. This material provides, besides the properties we've already seen for `THREE.MeshPhysicalMaterial`, the following properties to help you control what the material looks like:

- `clearCoat`: A value indicating a coating layer on top of the material. The higher this value is, the more coating is applied, and the more effective the `clearCoatRoughness` parameter is. This value ranges from `0` to `1` with a default of `0`.
- `clearCoatRoughness`: The roughness used for the coating of the material. The rougher it is, the more light is diffused. This is used together with the `clearCoat` property. This value ranges from `0` to `1` with a default of `0`.

As we've seen for other materials, it is quite hard to reason about the values you should use for your specific requirements. It's often the best choice to add a simple UI (as we do in the examples) and play around with the values to get to a combination that best reflects your needs. You can see this example in action by looking at the `mesh-physical-material.html` example:


localhost:8080/chapter-4/mesh-phy
localhost:8080/chapter-4/mesh-physical-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshPhysicalMaterial

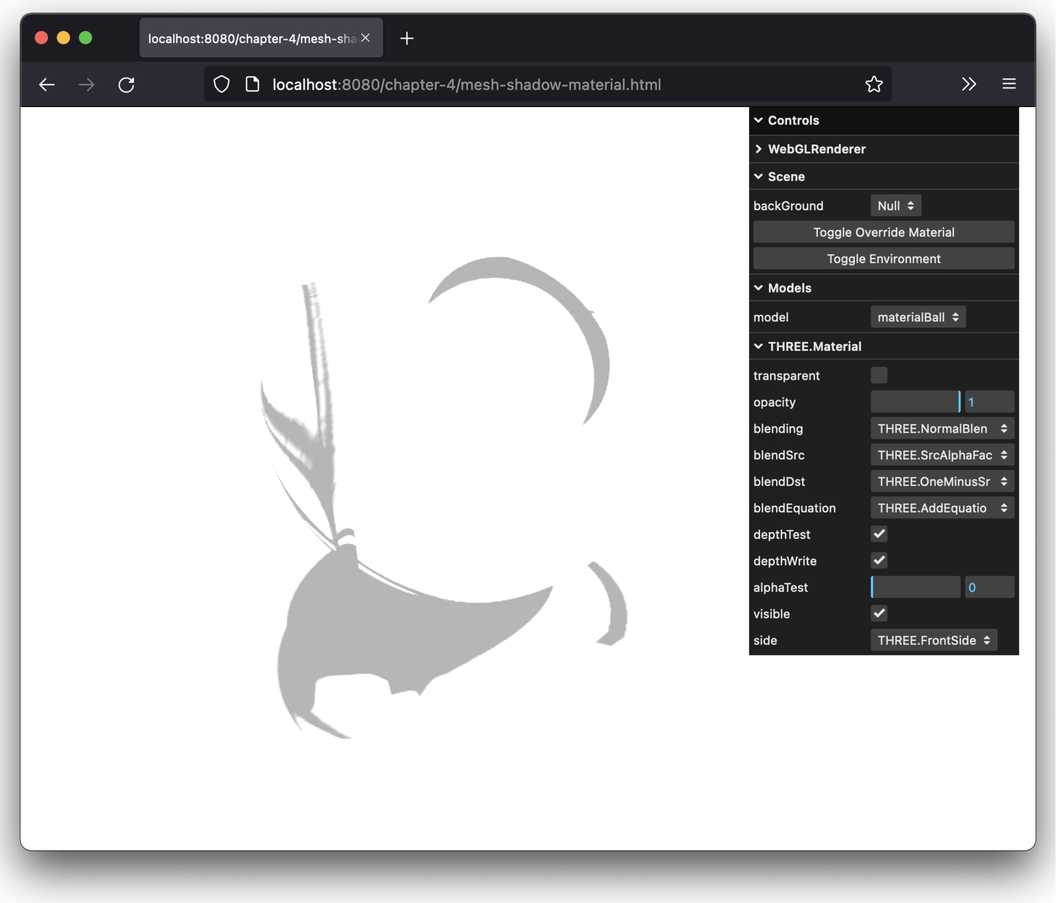
model knot
emissive ff2600
emissiveIntensity 0.309
roughness 0
metalness 1
clearcoat 1
clearcoatRoughness 0.229
color feffff
wireframe ☐
vertexColors ☐
envMaps reflection
map none
combine undefined
reflectivity 0.5
refractionRatio 0.98

Figure 4.18 – Mesh physical material using a clear coat to control reflection

Most of the advanced materials cast and receive shadows. The next material we're going to have a quick look at is a bit different than most. This material doesn't render the object itself, but only shows the shadows.

THREE.ShadowMaterial

`THREE.ShadowMaterial` is a special material that doesn't have any properties. You can't set the color or the shininess, or anything else. The only thing this material does is render the shadow the mesh would receive. The following screenshot should explain this:


localhost:8080/chapter-4/mesh-sha
localhost:8080/chapter-4/mesh-shadow-material.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material

backGround Null
model materialBall
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide

Figure 4.19 – Shadow material just rendering the shadows the mesh receives

Here, the only thing we can see are the shadows the object receives, nothing else. This material can, for instance, be combined with your own materials without you having to determine how to receive shadows.

The last of the advanced materials we'll explore is `THREE.ShaderMaterial`.

Using your own shaders with THREE.ShaderMaterial

`THREE.ShaderMaterial` is one of the most versatile and complex materials available in Three.js. With this material, you can pass in your own custom shaders that are directly run in the WebGL context. A shader converts Three.js JavaScript meshes into pixels on the screen. With these custom shaders, you can define exactly how your object should be rendered and how to override or alter the defaults from Three.js. In this section, we won't go into too much detail on how to write custom shaders; instead, we will just show you a couple of examples.

As we've already seen, `THREE.ShaderMaterial` has several properties you can set. With `THREE.ShaderMaterial`, Three.js passes in all the information regarding these properties to your custom shaders, but you still have to process the information to create colors and vertex positions. The following are the properties of `THREE.Material` that are passed into the shader, and that you can interpret for yourself:

- `wireframe`: This renders the material as a wireframe. This is great for debugging purposes.
- `shading`: This defines how shading is applied. The possible values are `THREE.SmoothShading` and `THREE.FlatShading`. This property isn't enabled in the example for this material. For an example, look at the *THREE. MeshNormalMaterial* section.
- `vertexColors`: You can define individual colors to be applied to each vertex with this property. Look at the `LineBasicMaterial` example in the *THREE.LineBasicMaterial* section, where we use this property to color the various parts of a line.
- `fog`: This determines whether this material is affected by global fog settings. This is not shown in action. If this is set to `false`, the global fog we saw in *Chapter 2* doesn't affect how this object is rendered.

Besides these properties that are passed into the shader, `THREE.ShaderMaterial` also provides several specific properties you can use to pass in additional information into your custom shader. Once again, we won't go into too much detail on how you can write your own shaders, since that would be a book on its own, so we'll just cover the basics:

- `fragmentShader`: This shader defines the color of each pixel that is passed in. Here, you need to pass in the string value of your fragment shader program.
- `vertextShader`: This shader allows you to change the position of each vertex that is passed in. Here, you need to pass in the string value of your vertex shader program.
- `uniforms`: This allows you to send information to your shader. The same information is sent to each vertex and fragment.

- `defines`: Converts custom key value pairs as `#define` code fragments. With these fragments, you can set some additional global variables in the shader programs or define your own custom global constants.
- `attributes`: These can change between each vertex and fragment. They are usually used to pass positional and normal-related data. If you want to use this property, you need to provide information for all the vertices of the geometry.
- `lights`: This determines whether light data should be passed into the shaders. This defaults to `false`.

Before we look at an example, we'll provide a quick explanation of the most important parts of `THREE.ShaderMaterial`. To work with this material, we have to pass in two different shaders:

- `vertexShader`: This is run on each vertex of the geometry. You can use this shader to transform the geometry by moving the position of the vertices around.
- `fragmentShader`: This is run on each fragment of the geometry. In `fragmentShader`, we return the color that should be shown for this specific fragment.

For all the materials we've discussed so far in this chapter, Three.js provides `fragmentShader` and `vertexShader`, so you don't have to worry about them and pass them in explicitly.

In this section, we'll look at a simple example that uses a very simple `vertexShader` program that changes the `x` and `y` coordinates of the vertices of a simple `THREE.PlainGeometry` and a `fragmentShader` program that changes the color based on some input.

Up next, you can see the complete code for our `vertexShader`. Note that writing shaders isn't done in JavaScript. You write shaders in a C-like language called GLSL (WebGL supports OpenGL ES Shading Language 1.0 — for more information on GLSL, see `https://www.khronos.org/webgl/`). The code for our simple shader looks like this:

```
uniform float time;
void main(){
  vec3 posChanged=position;
  posChanged.x=posChanged.x*(abs(sin(time*2.)));
  posChanged.y=posChanged.y*(abs(cos(time*1.)));
  posChanged.z=posChanged.z*(abs(sin(time*.5)));

  gl_Position=projectionMatrix*modelViewMatrix*vec4
    (posChanged,1.);
}
```

We won't go into too much detail here and just focus on the most important parts of this code. To communicate with the shaders from JavaScript, we use something called uniforms. In this example, we use the `uniform float time;` statement to pass in an external value.

Based on this value, we change the `x`, `y`, and `z` coordinates of the passed-in vertex (which is passed in as the position variable):

```
  posChanged.x=posChanged.x*(abs(sin(time*2.)));
  posChanged.y=posChanged.y*(abs(cos(time*1.)));
  posChanged.z=posChanged.z*(abs(sin(time*.5)));
```

The `posChanged` vector now contains the new coordinate for this vertex based on the passed-in time variable. The last step we need to perform is to pass this new position back to the renderer, which is always done like this for Three.js:

```
  gl_Position=projectionMatrix*modelViewMatrix*vec4
    (posChanged,1.);
```

The `gl_Position` variable is a special variable that is used to return the final position. This program is passed as a string value to the `vertexShader` property of `THREE.ShaderMaterial`. For `fragmentShader`, we do something similar. We've created a very simple fragment shader that just flips through colors based on the passed-in `time` uniform:

```
uniform float time;
void main(){

  float c1=mod(time,.5);
  float c2=mod(time,.7);
  float c3=mod(time,.9);
  gl_FragColor=vec4(c1,c2,c3,1.);
}
```

What we do in a `fragmentShader` is determine the color of a passed-in fragment (a pixel). The real shader programs take a lot into account, such as lights, the position of the vertex on the face, normals, and more. In this example, though, we just determine the `rgb` values of a color and return that in `gl_FragColor`, which is then shown on the final rendered mesh.

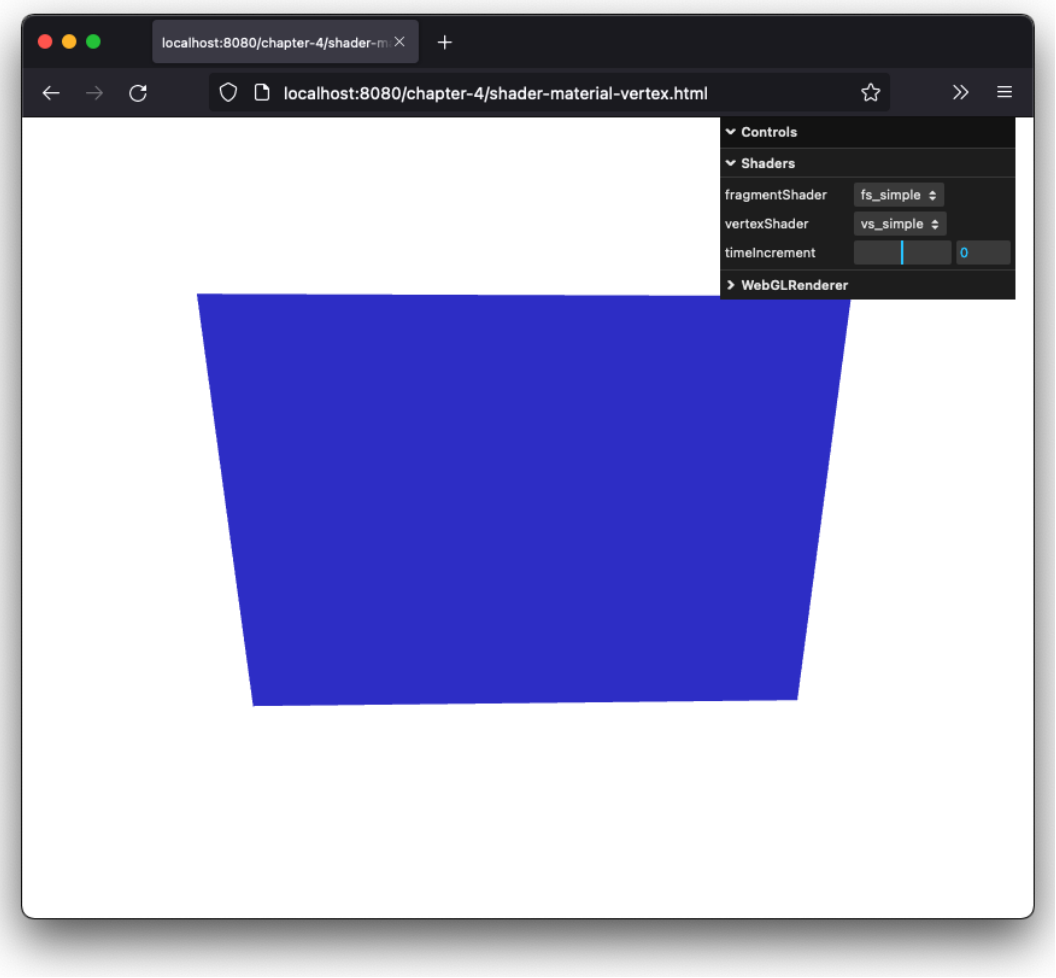
Now, we need to glue the geometry, the material, and the two shaders together. In Three.js, we can do that like this:

```
const geometry = new THREE.PlaneGeometry(10, 10, 100, 100)
const material = new THREE.ShaderMaterial({
  uniforms: {
    time: { value: 1.0 }
  },
  vertexShader: vs_simple,
  fragmentShader: fs_simple
})
const mesh = new THREE.Mesh(geometry, material)
```

Here, we define the `time` uniform, which will contain a value available in the shaders, and define `vertexShader` and `fragmentShader` as strings that we want to use. The only thing we need to do is make sure we change the `time` uniform in the render loop, and that's it:

```
// in the renderloop
material.uniforms.time.value += 0.005
```

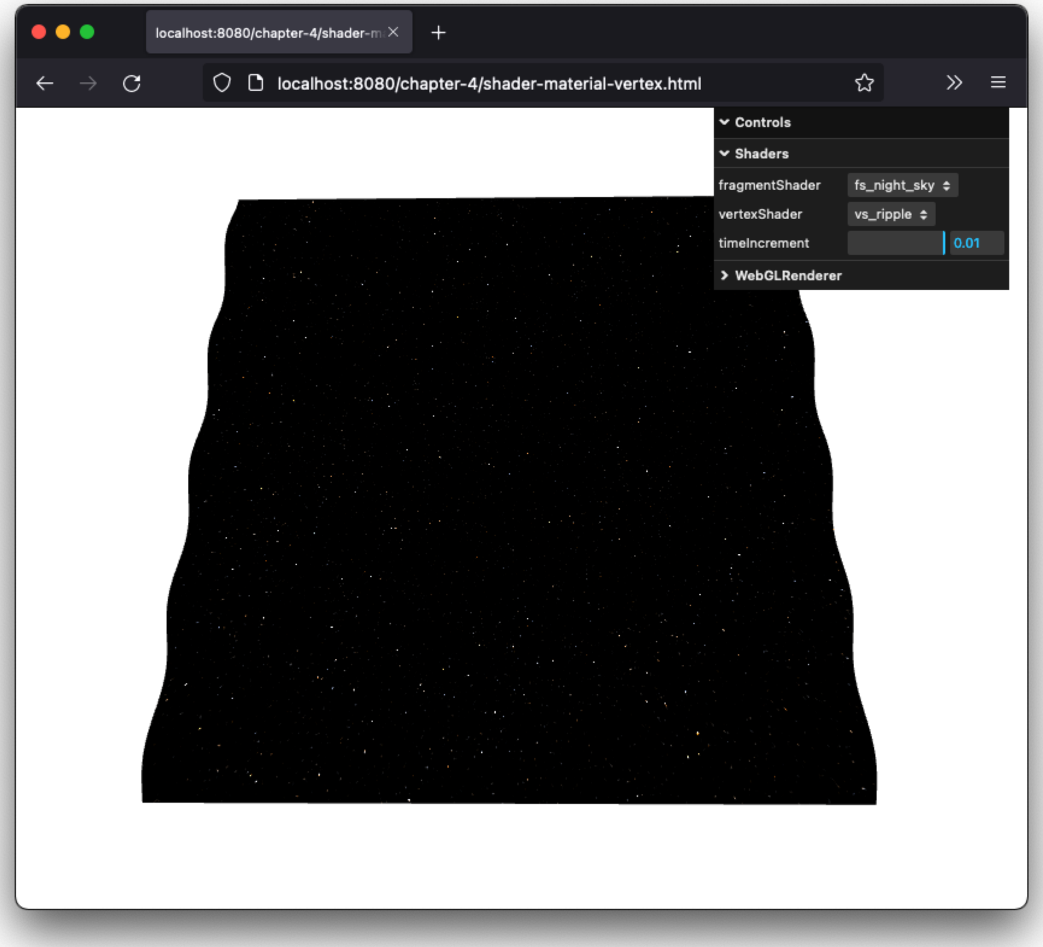
In the examples for this chapter, we've added a couple of simple shaders to experiment with. You will see the results if you open up the `shader-material-vertex.html` example in the `chapter-4` folder:


localhost:8080/chapter-4/shader-m
localhost:8080/chapter-4/shader-material-vertex.html
Controls
Shaders
WebGLRenderer

fragmentShader: fs_simple
vertexShader: vs_simple
timeIncrement: 0

Figure 4.20 – A shader material that shows a plane with the two example shader programs

In the dropdown menu, you can also find a couple of other shaders. For instance, the `fs_night_sky` fragment shader shows a starry night (based on the shader from `https://www.shadertoy.com/view/Nlffzj`). When combined with `vs_ripple`, you get a very nice-looking effect, completely running on the GPU, as shown here:


localhost:8080/chapter-4/shader-m
localhost:8080/chapter-4/shader-material-vertex.html
Controls
Shaders
WebGLRenderer

fragmentShader fs_night_sky
vertexShader vs_ripple
timeIncrement 0.01

Figure 4.21 – Rippling effect with a starry night fragment shader

It is possible to combine existing materials and reuse their fragment and vertex shaders with your own shaders. That way, you can, for instance, extend `THREE.MeshStandardMaterial` with some custom effects. Doing this in plain Three.js, however, is rather difficult to do and very error prone. Luckily, there is an open source project that provides us with a custom material that makes it very easy to wrap existing materials and add our own custom shaders. In the next section, we'll have a quick look at how that works.

Customizing existing shaders with CustomShaderMaterial

`THREE.CustomShader` doesn't come with the default Three.js distribution, but since we're using `yarn`, it is really easy to install (this is what you did when running the relevant commands from *Chapter 1*, *Creating Your First 3D Scene wth Three.js*). If you want more information on this module, you can check out `https://github.com/FarazzShaikh/THREE-CustomShaderMaterial`, where you can find documentation and additional examples.

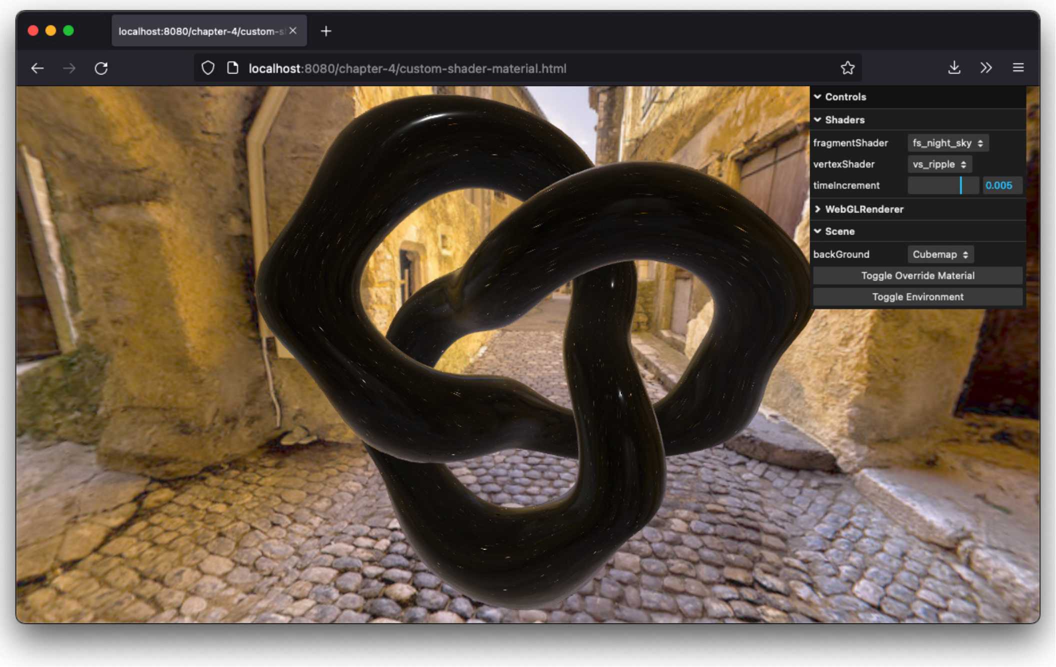
First, let's have a quick look at the code before we show some examples. Using `THREE.CustomShader` is the same as using the other materials:

```
const material = new CustomShaderMaterial({
  baseMaterial: THREE.MeshStandardMaterial,
  vertexShader: ...,
  fragmentShader: ...,
  uniforms: {
    time: { value: 0.2 },
    resolution: { value: new THREE.Vector2() }
  },
  flatShading: true,
  color: 0xffffff
})
```

As you can see, it is a bit of a combination of a normal material and a `THREE.ShaderMaterial`. The main thing to look at is the `baseMaterial` property. Here, you can add any of the standard Three.js materials. Any additional property you add, besides `vertexShader`, `fragmentShader`, and `uniforms`, gets applied to this `baseMaterial`. The `vertexshader`, `fragmentShader`, and `uniforms` properties work in the same way as we've seen for `THREE.ShaderMaterial`.

Out of the box, we need to make a couple of small changes to our shaders themselves. Recall the *Using your own shaders with THREE.ShaderMaterial* section, where we used `gl_Position` and `gl_FragColor` to set the final output of the position of a vertex and the color of a fragment. With this material, we use `csm_Position` for the final position and `csm_DiffuseColor` for the color. There are a couple more output variables you can use, which are explained in more detail here: `https://github.com/FarazzShaikh/THREE-CustomShaderMaterial#output-variables`.

If you open the `custom-shader-material` example, you will see how our simple shaders can be used together with the default materials from Three.js:



localhost:8080/chapter-4/custom-shader-material.html

fragmentShader fs_night_sky
vertexShader vs_ripple
timeIncrement 0.005
backGround Cubemap

Figure 4.22 – Ripple effect with an environment map using MeshStandardMaterials as the base

This approach gives you a relatively easy way to create your custom shaders, without having to start completely from scratch. You can just reuse the lights and shadow effects from the default shaders and extend them with your custom-needed functionality.

So far, we've looked at materials that work with meshes. Three.js also provides materials that can be used together with line geometries. In the next section, we'll explore those materials.

Materials you can use for a line geometry

The last couple of materials we're going to look at can only be used on one specific mesh: `THREE.Line`. As the name implies, this is just a single line that only consists of lines and doesn't contain any faces. Three.js provides two different materials you can use on a `THREE.Line` geometry, as follows:

- `THREE.LineBasicMaterial`: This basic material for a line allows you to set the `color` and `vertexColors` properties.

- `THREE.LineDashedMaterial`: This has the same properties as `THREE.LineBasicMaterial` but allows you to create a dashed line effect by specifying `dash` and `spacing` sizes.

We'll start with the basic variant; after that, we'll look at the dashed variant.

THREE.LineBasicMaterial

The materials that are available for the `THREE.Line` geometry are very simple. It inherits all of the properties from `THREE.Material`, but the following are the properties that are most important for this material:

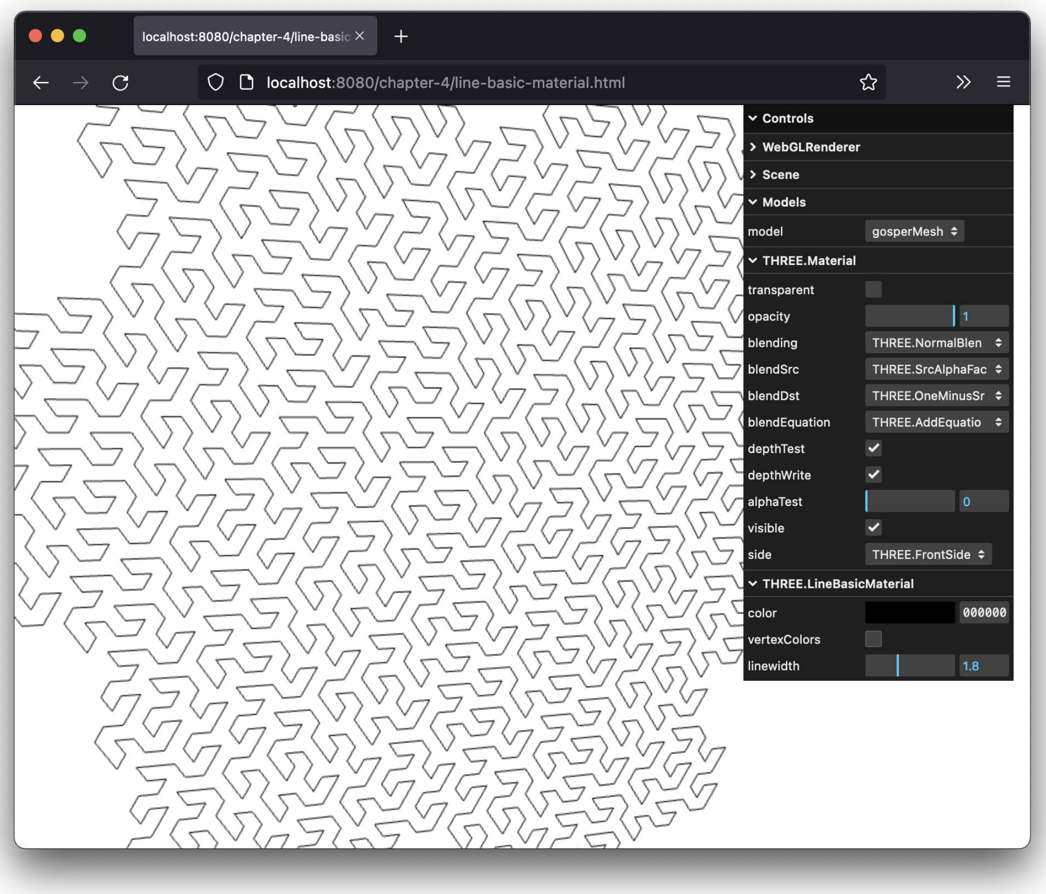
- `color`: This determines the color of the line. If you specify `vertexColors`, this property is ignored. An example of how to do this is shown in the following code fragment.
- `vertexColors`: You can supply a specific color for each vertex by setting this property to the `THREE.VertexColors` value.

Before we look at an example of `THREE.LineBasicMaterial`, let's have a quick look at how we can create a `THREE.Line` mesh from a set of vertices and combine that with `THREE.LineMaterial` to create the mesh, as shown in the following code:

```
const points = gosper(4, 50)
const lineGeometry = new THREE.BufferGeometry().
  setFromPoints(points)
const colors = new Float32Array(points.length * 3)
points.forEach((e, i) => {
  const color = new THREE.Color(0xffffff)
  color.setHSL(e.x / 100 + 0.2, (e.y * 20) / 300, 0.8)
  colors[i * 3] = color.r
  colors[i * 3 + 1] = color.g
  colors[i * 3 + 2] = color.b
})
lineGeometry.setAttribute('color', new THREE.
  BufferAttribute(colors, 3, true))
const material = new THREE.LineBasicMaterial(0xff0000);
const mesh = new THREE.Line(lineGeometry, material)
mesh.computeLineDistances()
```

The first part of this code fragment, `const points = gosper(4, 60)`, is used as an example to get a set of `x`, `y`, and `z` coordinates. This function returns a Gosper curve (for more information, check out `https://mathworld.wolfram.com/Peano-GosperCurve.html`), which is a simple algorithm that fills a 2D space. What we do next is create a `THREE.BufferGeometry` instance and call the `setFromPoints` function to add the generated points. For each coordinate, we also calculate a color value that we use to set the `color` attribute of the geometry. Note `mesh.computeLineDistances` at the end of this code fragment. This is needed when you want to have dashed lines when using `THREE.LineDashedMaterial`.

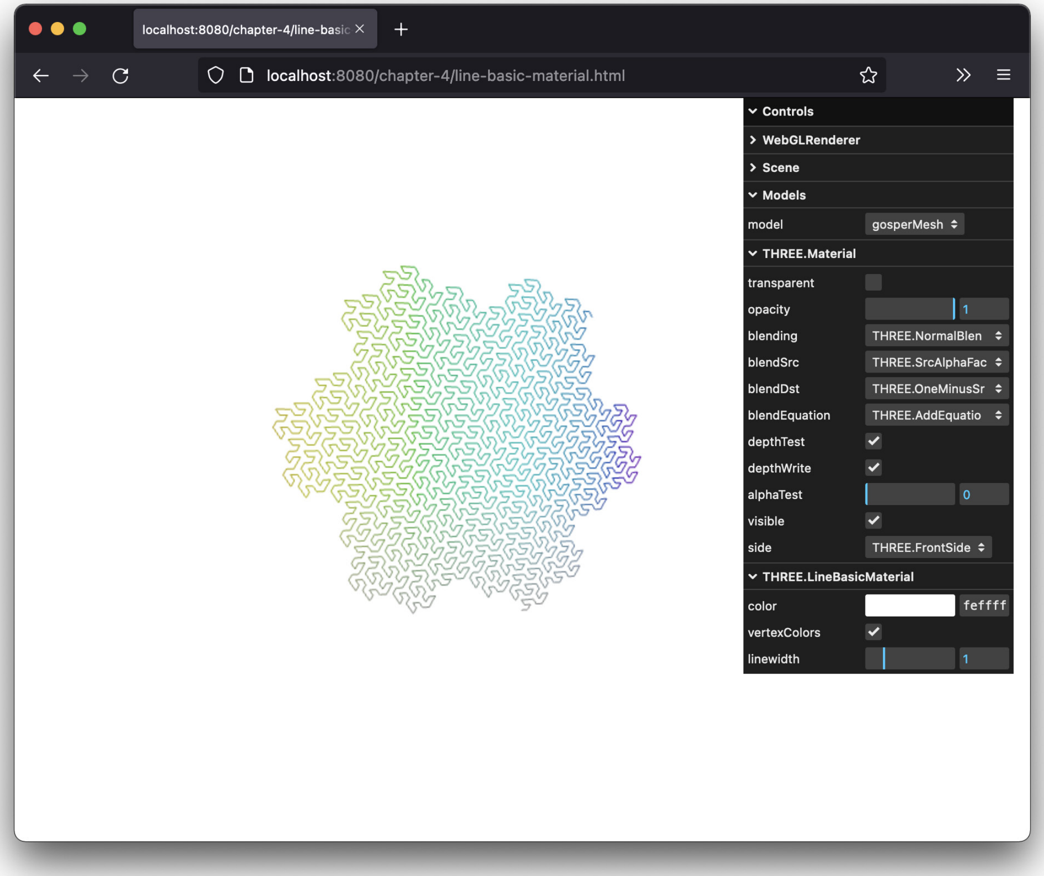
Now that we have our geometry, we can create `THREE.LineBasicMaterial` and use this together with the geometry to create a `THREE.Line` mesh. You can see the result in the `line-basic-material.html` example. The following screenshot shows this example:


localhost:8080/chapter-4/line-basic
localhost:8080/chapter-4/line-basic-material.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.LineBasicMaterial

model gosperMesh
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color 000000
vertexColors ☐
linewidth 1.8

Figure 4.23 – Line basic material

This is a line geometry created with `THREE.LineBasicMaterial`. If we enable the `vertexColors` property, we will see that the individual line segments are colored:



localhost:8080/chapter-4/line-basic-material.html

Controls
WebGLRenderer
Scene
Models
model gosperMesh
THREE.Material
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
THREE.LineBasicMaterial
color feffff
vertexColors ☑
linewidth 1

Figure 4.24 – Line basic material with vertex colors

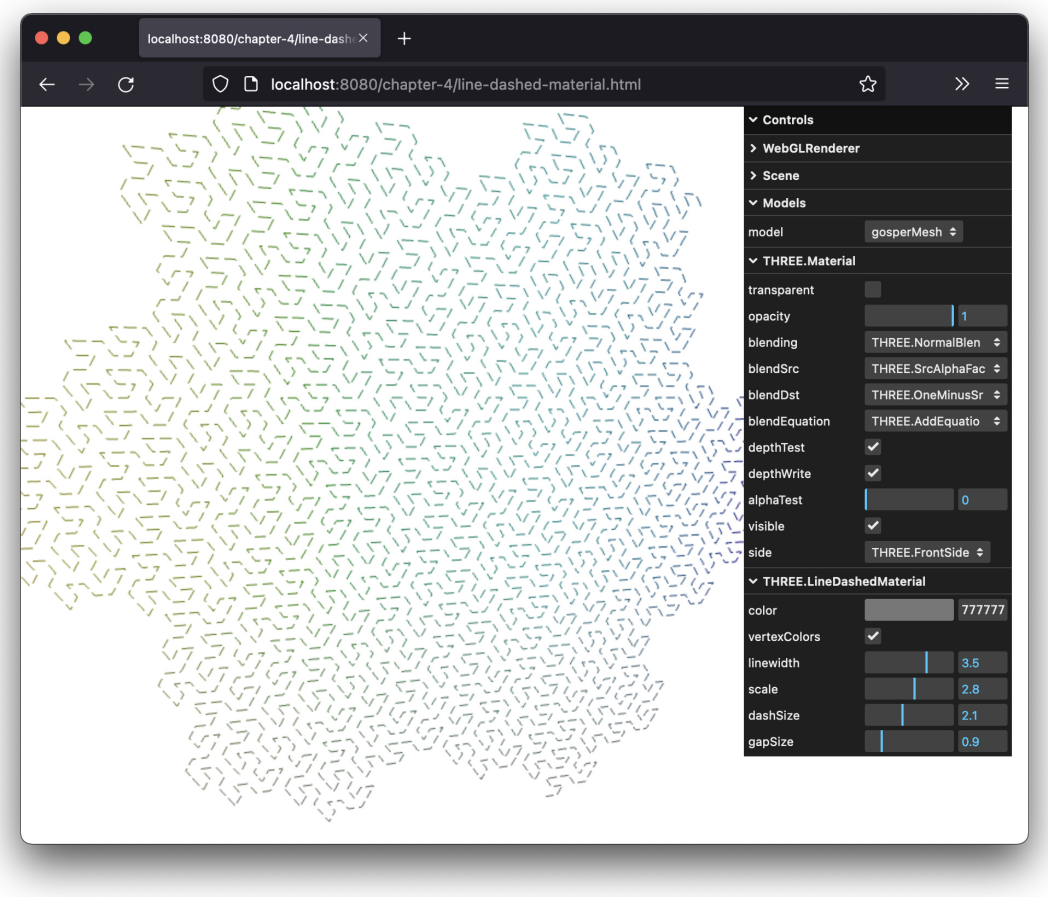
The next and last material we'll discuss in this chapter is only slightly different from `THREE.LineBasicMaterial`. With `THREE.LineDashedMaterial`, not only can we color lines, but we can also add spaces to those lines.

THREE.LineDashedMaterial

This material has the same properties as `THREE.LineBasicMaterial` and three additional ones you can use to define the dash width and the width of the gaps between the dashes:

- `scale`: This scales `dashSize` and `gapSize`. If the scale is smaller than `1`, `dashSize` and `gapSize` increase, whereas if the scale is larger than `1`, `dashSize` and `gapSize` decrease.
- `dashSize`: This is the size of the dash.
- `gapSize`: This is the size of the gap.

This material works almost exactly like `THREE.LineBasicMaterial`. The only difference is that you have to call `computeLineDistances()` (which is used to determine the distance between the vertices that make up a line). If you don't do this, the gaps won't be shown correctly. An example of this material can be found in `line-dashed-material.html` and looks like this:



localhost:8080/chapter-4/line-dashed-material.html

Controls
WebGLRenderer
Scene
Models
model: gosperMesh
THREE.Material
transparent: ☐
opacity: 1
blending: THREE.NormalBlen
blendSrc: THREE.SrcAlphaFac
blendDst: THREE.OneMinusSr
blendEquation: THREE.AddEquatio
depthTest: ☑
depthWrite: ☑
alphaTest: 0
visible: ☑
side: THREE.FrontSide
THREE.LineDashedMaterial
color: 777777
vertexColors: ☑
linewidth: 3.5
scale: 2.8
dashSize: 2.1
gapSize: 0.9

Figure 4.25 – A Gosper mesh with the line dashed material

That's it for this section on materials used for lines. You've seen that Three.js only provides a few materials specifically for line geometries, but with these materials, especially in combination with `vertexColors`, you should be able to style the line geometries any way you want.

Summary

Three.js gives you a lot of materials you can use to skin your geometries. The materials range from the very simple (`THREE.MeshBasicMaterial`) to the complex (`THREE.ShaderMaterial`), where you can provide your own `vertexShader` and `fragmentShader` programs. Materials share a lot of basic properties. If you know how to use a single material, you'll probably also know how to use the other materials. Note that not all materials respond to the lights in your scene. If you want a material that takes lighting into effect, you can usually just use `THREE.MeshStandardMaterial`. If you need more control, you can also look at `THREE.MeshPhysicalMaterial`, `THREE.MeshPhongMaterial`, or `THREE.MeshLamberMaterial`. Determining the effect of certain material properties from just code is very hard. Often, a good idea is to use the control GUI approach to experiment with these properties, as we showed in this chapter.

Also, remember that most of the properties of a material can be modified at runtime. Some, though (for example, `side`), can't be modified at runtime. If you change such a value, you need to set the `needsUpdate` property to `true`. For a complete overview of what can and cannot be changed at runtime, see the following page: `https://threejs.org/docs/#manual/en/introduction/How-to-update-things`.

In this and the previous chapters, we talked about geometries. We used these in our examples and explored a couple of them. In the next chapter, you'll learn everything about geometries and how you can work with them.

5

Learning to Work with Geometries

In the previous chapters, you learned a lot about how to work with Three.js. Now you know how to create a basic scene, add lighting, and configure the material for your meshes. In *Chapter 2*, *The Basic Components That Make Up a Three.js Scene*, we touched upon (but didn't really go into) the details of the available geometries that Three.js provides and that you can use to create your 3D objects. In this chapter and *Chapter 6*, *Exploring Advanced Geometries*, we'll walk you through all the geometries that Three.js provides out of the box (except `THREE.Line`, which we discussed in *Chapter 4*, *Working with Three.js Materials*).

In this chapter, we'll look at the following geometries:

- `THREE.CircleGeometry`
- `THREE.RingGeometry`
- `THREE.PlaneGeometry`
- `THREE.ShapeGeometry`
- `THREE.BoxGeometry`
- `THREE.SphereGeometry`
- `THREE.CylinderGeometry`
- `THREE.ConeGeometry`
- `THREE.TorusGeometry`
- `THREE.TorusKnotGeometry`
- `THREE.PolyhedronGeometry`
- `THREE.IcosahedronGeometry`
- `THREE.OctahedronGeometry`

- `THREE.TetraHedronGeometry`
- `THREE.DodecahedronGeometry`

Before we look into the geometries provided by Three.js, we'll first look a bit deeper into how Three.js represents geometries internally as `THREE.BufferGeometry`. In some documentation, you might still encounter `THREE.Geometry` as a base object for all geometries. This has been replaced completely in the newer version by `THREE.BufferGeometry`, which generally provides better performance, since it can easily get its data to the GPU. It is, however, a bit more difficult to use than the old `THREE.Geometry`.

With `THREE.BufferGeometry`, all properties of geometry are identified by a set of attributes. An attribute is basically an array with some additional metadata that contains information about the position of the vertices. Attributes are also used to store additional information about a vertex – for example, its color. To use an attribute to define a vertex and a face, you use the following two properties of `THREE.BufferGeometry`:

- `attributes`: The `attributes` properties are used to store information that can be directly passed to the GPU. For instance, for defining a shape, you define a `Float32Array`, where every three values define the position of a vertex.

  Each of the three vertices is then interpreted as a face. This can be defined in `THREE.BufferGeometry` like this: `geometry.setAttribute( 'position', new THREE.BufferAttribute( arrayOfVertices, 3 ) );`.
- `index`: By default, faces don't need to be explicitly defined (every three consecutive positions are interpreted as a single face), but using the `index` property, we can explicitly define which vertices together form a face: `geometry.setIndex( indicesArray );`.

For using the geometries in this chapter, you don't need to think about these internal attributes, since Three.js takes care of setting them correctly when you construct a geometry. If you want to create a geometry from scratch, though, you need to use the properties shown in the previous list.

In Three.js, we have a couple of geometries that result in a 2D mesh, and a larger number of geometries that create a 3D mesh. In this chapter, we will discuss the following topics:

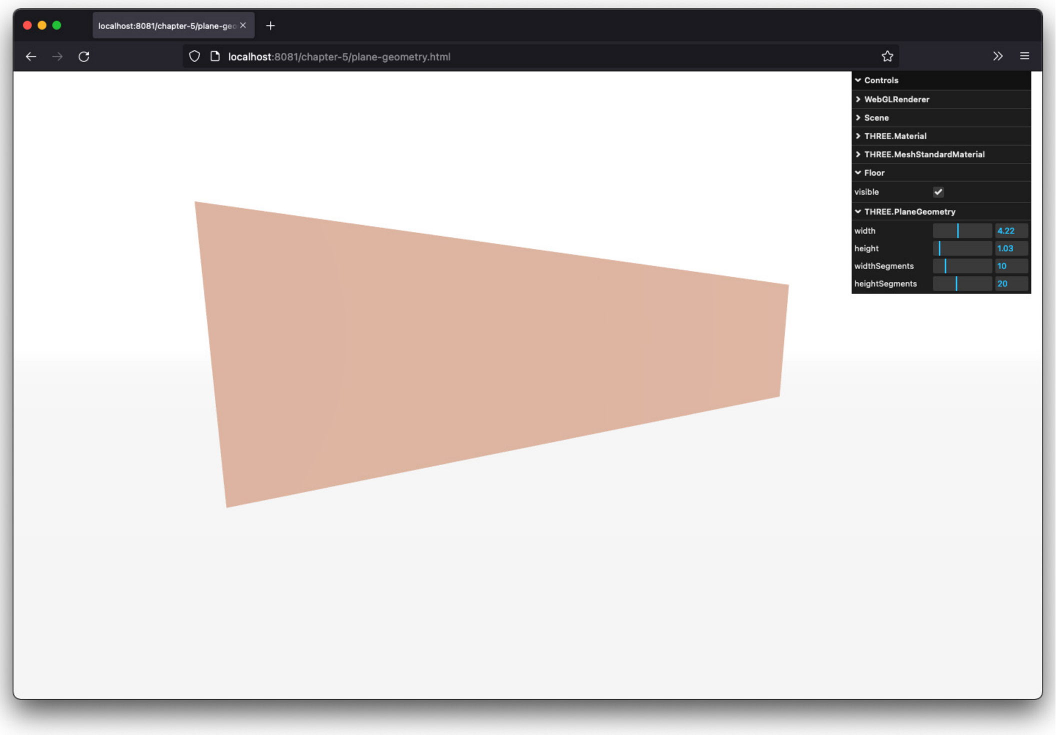
- 2D geometries
- 3D geometries

2D geometries

2D objects look like flat objects and, as the name implies, only have two dimensions. In this section, we'll first look at the 2D geometries: `THREE.CircleGeometry`, `THREE.RingGeometry`, `THREE.PlaneGeometry`, and `THREE.ShapeGeometry`.

THREE.PlaneGeometry

A `THREE.PlaneGeometry` object can be used to create a very simple 2D rectangle. For an example of this geometry, look at the `plane-geometry.html` example in the sources for this chapter. A rectangle that was created using `THREE.PlaneGeometry` is shown in the following screenshot:

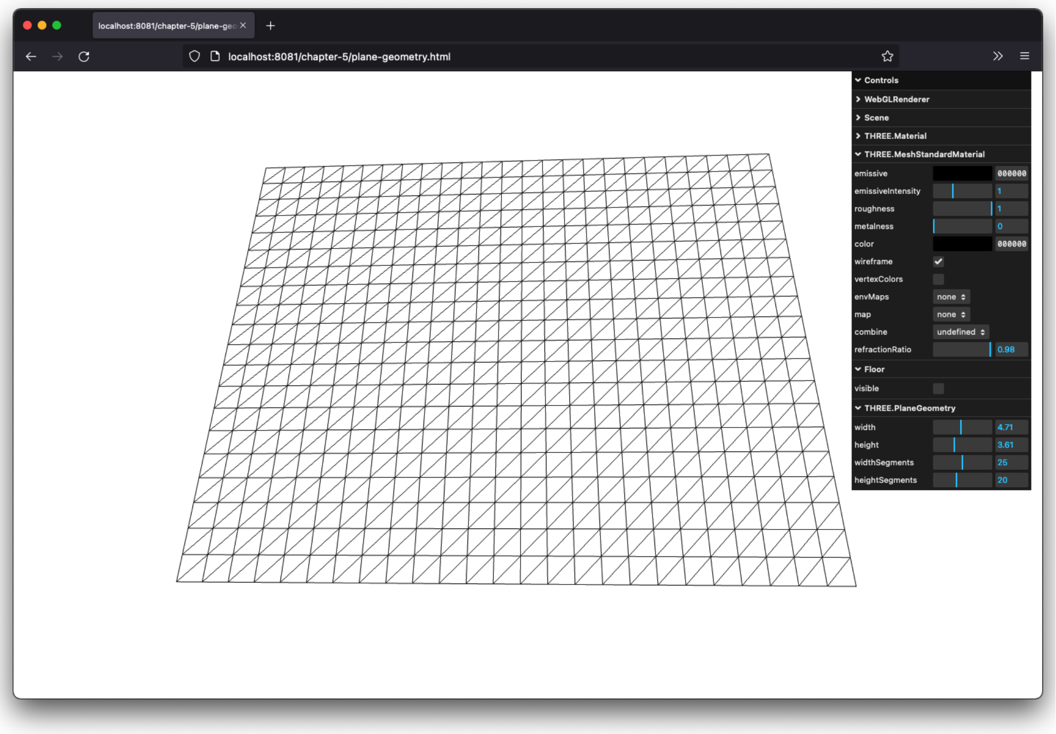

localhost:8081/chapter-5/plane-geo
localhost:8081/chapter-5/plane-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PlaneGeometry

visible ☑

width 4.22
height 1.03
widthSegments 10
heightSegments 20

Figure 5.1 – Plane geometry

In the examples for this chapter, we've added a control GUI that you can use to control the properties of the geometry (in this case, `width`, `height`, `widthSegments`, and `heightSegments`), and also change the material (and its properties), disable shadows, and hide the ground plane. For instance, if you want to see the individual faces of this shape, you can easily show them by disabling the ground plane and enabling the `wireframe` property of the selected material:


localhost:8081/chapter-5/plane-ge
localhost:8081/chapter-5/plane-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PlaneGeometry

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☐
width 4.71
height 3.61
widthSegments 25
heightSegments 20

Figure 5.2 – Plane geometry as a wireframe

Creating a `THREE.PlaneGeometry` object is very simple, and is done as follows:

```
new THREE.PlaneGeometry(width, height, widthSegments,
heightSegments)
```

In this example for `THREE.PlaneGeometry`, you can change these properties and directly see the effect it has on the resulting 3D object. An explanation of these properties is shown in the following list:

- `width`: This is the width of the rectangle.
- `height`: This is the height of the rectangle.
- `widthSegments`: This is the number of segments that the width should be divided into. This defaults to 1.
- `heightSegments`: This is the number of segments the height should be divided into. This defaults to 1.

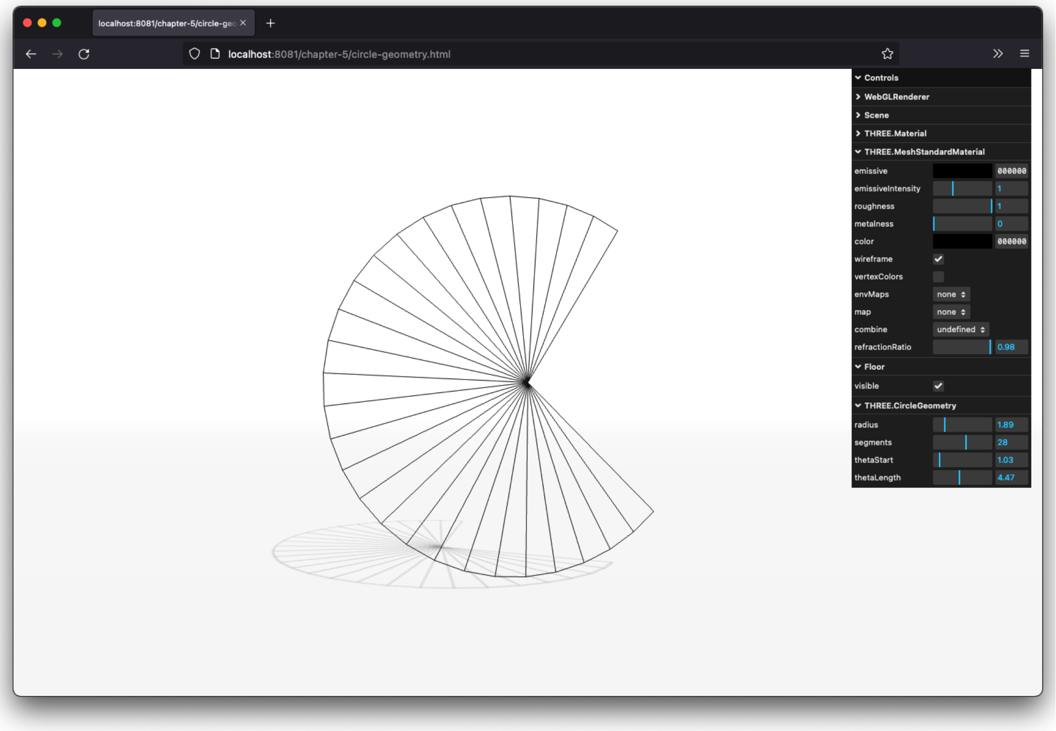
As you can see, this is not a very complex geometry. You just specify the size, and you're done. If you want to create more faces (for example, when you want to create a checkered pattern), you can use the `widthSegments` and `heightSegments` properties to divide the geometry into smaller faces.

> **Note**
>
> If you want to access the properties of a geometry after it has been created, you can't just say `plane.width`. To access the properties of a geometry, you have to use the `parameters` property of the object. So, to get the `width` property of the plane object we created in this section, you'd have to use `plane.parameters.width`.

THREE.CircleGeometry

You can probably already guess what `THREE.CircleGeometry` creates. With this geometry, you can create a very simple 2D circle (or part of a circle). First, let's look at the example for this geometry, `circle-geometry.html`.

In the following screenshot, you can find an example where we created a `THREE.CircleGeometry` object with a `thetaLength` value that is smaller than `2 * PI`:


localhost:8081/chapter-5/circle-geo
localhost:8081/chapter-5/circle-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98

Floor

visible ☑

THREE.CircleGeometry

radius 1.89
segments 28
thetaStart 1.03
thetaLength 4.47

Figure 5.3 – 2D circle geometry

In this example, you can see and control a mesh that's been created by using `THREE.CircleGeometry`. `2 * PI` represents a complete circle in radians. If you'd rather work with degrees than radians, converting between them is very easy.

The following two functions can help you to convert between radians and degrees, as follows:

```
const deg2rad = (degrees) => (degrees * Math.PI) / 180
const rad2deg = (radians) => (radians * 180) / Math.PI
```

When you create `THREE.CircleGeometry`, you can specify a few properties that define what the circle looks like, as follows:

- `radius`: The radius of a circle defines its size. The radius is the distance from the center of the circle to its edge. The default value is 50.
- `segments`: This property defines the number of faces that are used to create the circle. The minimum number is 3, and if not specified, this number defaults to 8. A higher value means a smoother circle.
- `thetaStart`: This property defines the position from which to start drawing the circle. This value can range from 0 to `2 * PI`, and the default value is 0.
- `thetaLength`: This property defines to what extent the circle is complete. This defaults to `2 * PI` (a full circle) when not specified. For instance, if you specify `0.5 * PI` for this value, you'll get a quarter circle. Use this property together with the `thetaStart` property to define the shape of a circle.

You can create a full circle by just specifying the `radius` and the `segments`:

```
new THREE.CircleGeometry(3, 12)
```

If you wanted to create half a circle from this geometry, you'd use something like this:

```
new THREE.CircleGeometry(3, 12, 0, Math.PI);
```

This creates a circle with a `radius` of `3` that is split into `12` segments. The circle starts at a default of `0` and is only drawn halfway since we specify `thetaLength` as `Math.PI`, which is half a circle.

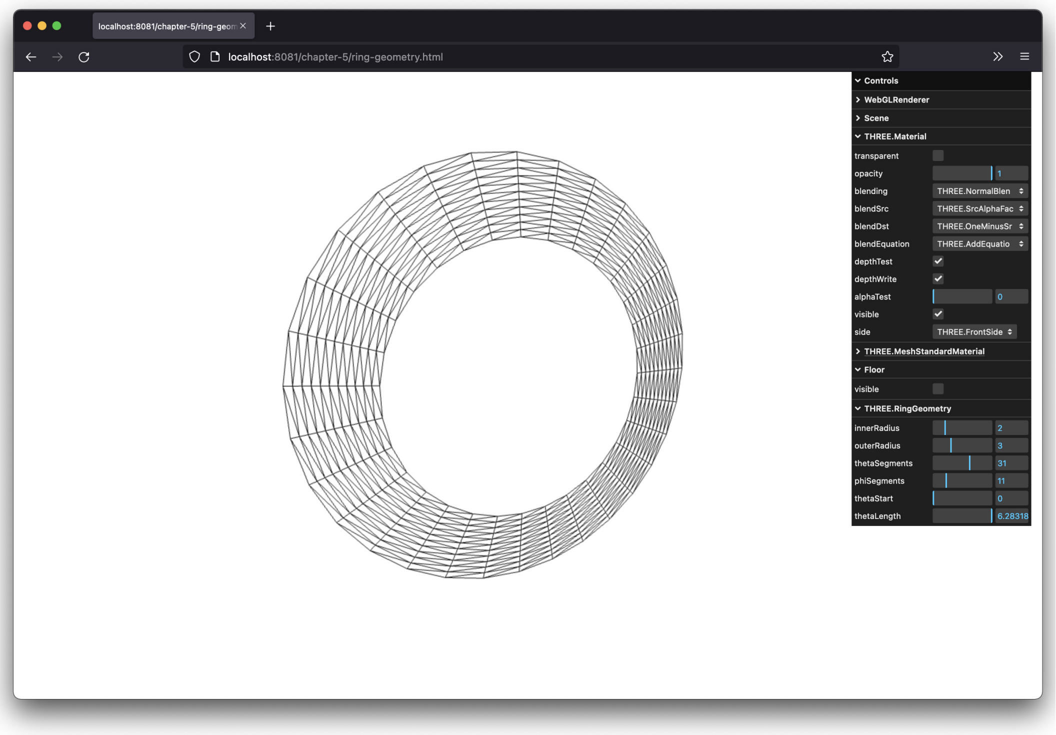
Before moving on to the next geometry, here's a quick note on the orientation that Three.js uses when creating these 2D shapes (`THREE.PlaneGeometry`, `THREE.CircleGeometry`, `THREE.RingGeometry`, and `THREE.ShapeGeometry`): Three.js creates these objects standing up, so they are aligned along the *x-y* plane. This is very logical since they are 2D shapes. However, often, especially with `THREE.PlaneGeometry`, you want to have the mesh lying down on the ground (the *x*-*z* plane), as some sort of ground area on which you can position the rest of your objects. The easiest way to create a 2D object that is horizontally orientated instead of vertical is by rotating the mesh a quarter rotation backward (`-PI/2`) around its *x*-axis, as follows:

```
mesh.rotation.x =- Math.PI/2;
```

That's all for `THREE.CircleGeometry`. The next geometry, `THREE.RingGeometry`, looks a lot like `THREE.CircleGeometry`.

THREE.RingGeometry

With `THREE.RingGeometry`, you can create a 2D object that not only closely resembles `THREE.CircleGeometry`, but also allows you to define a hole in the center (see `ring-geometry.html`):


localhost:8081/chapter-5/ring-geom
localhost:8081/chapter-5/ring-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.RingGeometry

transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide

visible ☐

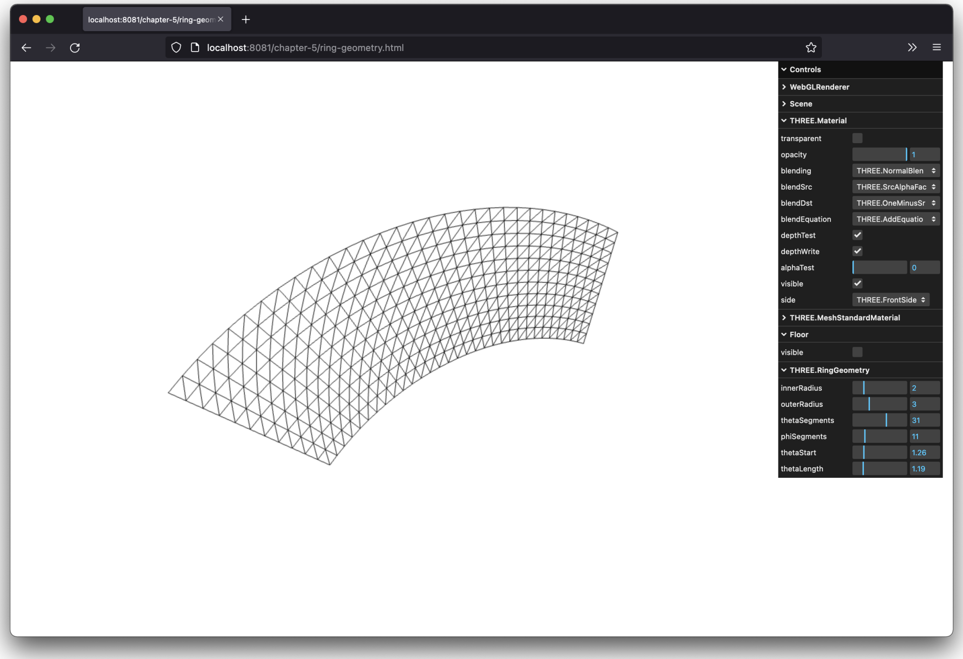
innerRadius 2
outerRadius 3
thetaSegments 31
phiSegments 11
thetaStart 0
thetaLength 6.28318

Figure 5.4 – 2D ring geometry

When creating a `THREE.RingGeometry` object, you can use the following properties:

- `innerRadius`: The inner radius of a circle defines the size of the center hole. If this property is set to `0`, no hole will be shown. The default value is 0.
- `outerRadius`: The outer radius of a circle defines its size. The radius is the distance from the center of the circle to its edge. The default value is 50.
- `thetaSegments`: This is the number of diagonal segments that will be used to create the circle. A higher value means a smoother ring. The default value is 8.
- `phiSegments`: This is the number of segments required to be used along the length of the ring. The default value is 8. This doesn't really affect the smoothness of the circle but increases the number of faces.
- `thetaStart`: This defines the position from which to start drawing the circle. This value can range from 0 to `2 * PI`, and the default value is 0.
- `thetaLength`: This defines the extent to which the circle is complete. This defaults to `2 * PI` (a full circle) when not specified. For instance, if you specify `0.5 * PI` for this value, you'll get a quarter circle. Use this property together with the `thetaStart` property to define the shape of the ring.

The following screenshot shows how `thetaStart` and `thetaLength` work with `THREE.RingGeometry`:


localhost:8081/chapter-5/ring-geom
localhost:8081/chapter-5/ring-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.RingGeometry

transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide

visible ☐

innerRadius 2
outerRadius 3
thetaSegments 31
phiSegments 11
thetaStart 1.26
thetaLength 1.19

Figure 5.5 – 2D ring geometry with different theta

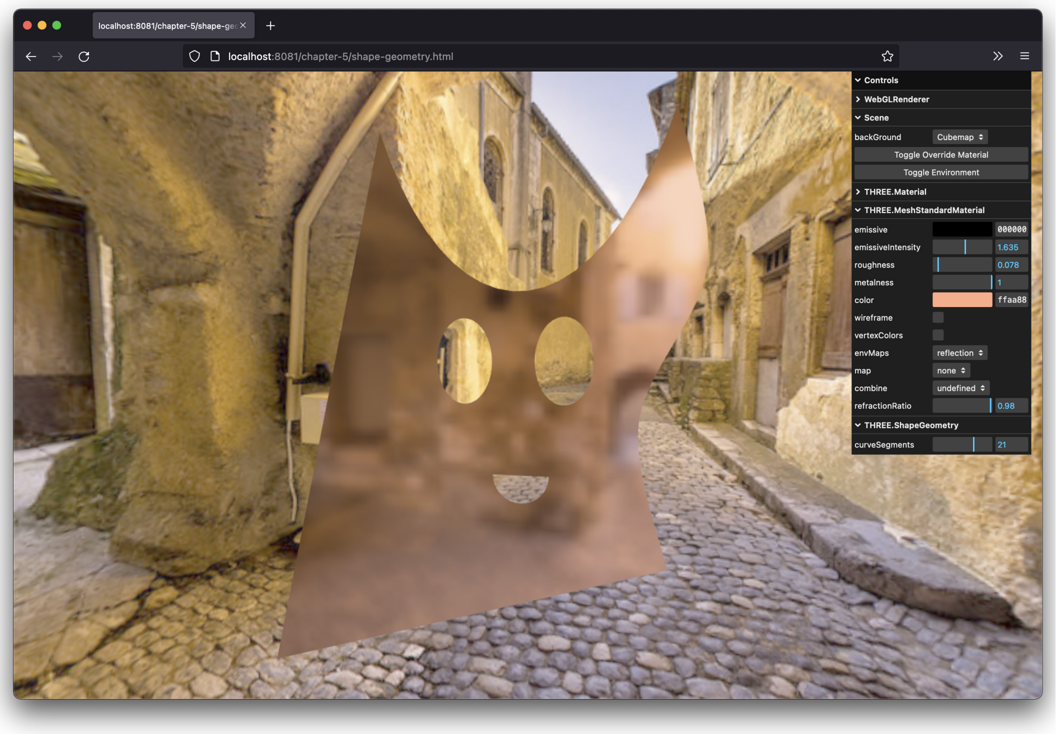
In the next section, we'll look at the last of the 2D shapes: `THREE.ShapeGeometry`.

THREE.ShapeGeometry

`THREE.PlaneGeometry` and `THREE.CircleGeometry` have limited ways of customizing their appearance. If you want to create custom 2D shapes, you can use `THREE.ShapeGeometry`.

With `THREE.ShapeGeometry`, you have a couple of functions you can call to create your own shapes. You can compare this functionality with the `<path/>` element functionality, which is also available to the HTML canvas element and SVG. Let's start with an example and after that, we'll show you how you can use the various functions to draw your own shape. The `shape-geometry.html` example can be found in the sources of this chapter.

The following screenshot shows this example:


localhost:8081/chapter-5/shape-ge
localhost:8081/chapter-5/shape-geometry.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
THREE.Material
THREE.MeshStandardMaterial
THREE.ShapeGeometry

backGround: Cubemap
emissive: 000000
emissiveIntensity: 1.635
roughness: 0.078
metalness: 1
color: ffaa88
wireframe: ☐
vertexColors: ☐
envMaps: reflection
map: none
combine: undefined
refractionRatio: 0.98
curveSegments: 21

Figure 5.6 – A custom shape geometry

In this example, you can see a custom-created 2D shape. Before going into the description of the properties, first, let's look at the code that is used to create this shape. Before we create a `THREE.ShapeGeometry` object, we first have to create a `THREE.Shape` object. You can trace these steps by looking at the previous screenshot, where we start in the bottom-right corner. Here's how we created a `THREE.Shape` object:

```
const drawShape = () => {
  // create a basic shape
  const shape = new THREE.Shape()
  // startpoint
  // straight line upwards
  shape.lineTo(10, 40)
  // the top of the figure, curve to the right
  shape.bezierCurveTo(15, 25, 25, 25, 30, 40)
  // spline back down
  shape.splineThru([new THREE.Vector2(32, 30), new
    THREE.Vector2(28, 20), new THREE.Vector2(30, 10)])
  // add 'eye' hole one
  const hole1 = new THREE.Path()
  hole1.absellipse(16, 24, 2, 3, 0, Math.PI * 2, true)
  shape.holes.push(hole1)
  // add 'eye hole 2'
  const hole2 = new THREE.Path()
  hole2.absellipse(23, 24, 2, 3, 0, Math.PI * 2, true)
  shape.holes.push(hole2)
  // add 'mouth'
  const hole3 = new THREE.Path()
  hole3.absarc(20, 16, 2, 0, Math.PI, true)
  shape.holes.push(hole3)
  return shape
}
```

In this piece of code, you can see that we created the outline of this shape using lines, curves, and splines. After that, we punched a number of holes into this shape by using the `holes` property of `THREE.Shape`.

In this section, though, we're talking about `THREE.ShapeGeometry` and not `THREE.Shape` – to create a geometry from `THREE.Shape`, we need to pass in `THREE.Shape` (returned, in our case, from the `drawShape()` function) as the argument to `THREE.ShapeGeometry`. You can also pass in an array of a `THREE.Shape` object, but in our example, we only use one object:

```
new THREE.ShapeGeometry(drawShape())
```

The result of this function is a geometry that can be used to create a mesh. Besides the shape you want to convert into `THREE.ShapeGeometry`, you can also pass in a number of additional options objects as the second argument:

- `curveSegments`: This property determines how smooth the curves created from the shape are. The default value is 12.
- `material`: This is the `materialIndex` property that is used for the faces that are created for the specified shapes. So, if you pass in multiple materials, you can specify which of those materials should be applied to the faces of the shape that is created.
- `UVGenerator`: When you use a texture with your material, the UV mapping determines what part of a texture is used for a specific face. With the `UVGenerator` property, you can pass in your own object, which will create the UV settings for the faces that are created for the shapes passed in. More information on UV settings can be found in *Chapter 10*, *Loading and Working with Textures*. If none are specified, `THREE.ExtrudeGeometry.WorldUVGenerator` is used.

The most important part of `THREE.ShapeGeometry` is `THREE.Shape`, which you use to create the shape, so let's look at the list of drawing functions you can use to create `THREE.Shape`:

- `moveTo(x,y)`: Move the drawing position to the `x-` and `y-`coordinates that are specified.
- `lineTo(x,y)`: Draw a line from the current position (for example, set by the `moveTo` function) to the `x-` and `y-`coordinates that have been provided.
- `quadraticCurveTo(aCPx, aCPy, x, y)`: There are two different ways of specifying curves. You can use the `quadraticCurveTo` function, or you can use the `bezierCurveTo` function. The difference between these two functions is how you specify the curvature of the curve.

  For a quadratic curve, we need to specify one additional point (using the `aCPx` and `aCPy` arguments), and the curve is based solely on that point and, of course, the specified endpoint (from the `x` and `y` arguments). For a cubic curve (used by the `bezierCurveTo` function), you specify two additional points to define the curve. The starting point is the current position of the path.

The following diagram explains the differences between these two options:

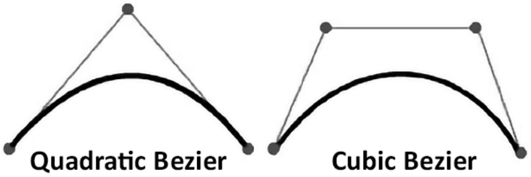
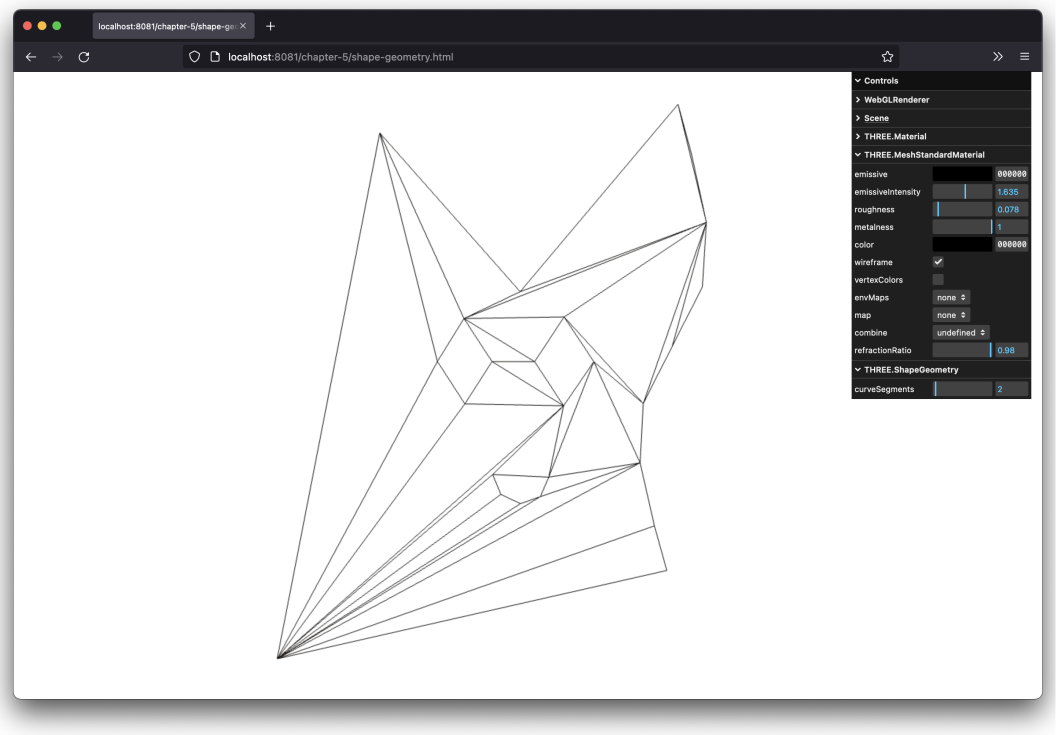


Figure 5.7 – Quadratic bezier and cubic bezier

- `bezierCurveTo(aCPx1, aCPy1, aCPx2, aCPy2, x, y)`: This draws a curve based on the arguments supplied. For an explanation, see the previous list entry. The curve is drawn based on the two coordinates that define the curve (`aCPx1`, `aCPy1`, `aCPx2`, and `aCPy2`) and the end coordinates (`x` and `y`). The starting point is the current position of the path.
- `splineThru(pts)`: This function draws a fluid line through the set of coordinates provided, (`pts`). This argument should be an array of `THREE.Vector2` objects. The starting point is the current position of the path.
- `arc(aX, aY, aRadius, aStartAngle, aEndAngle, aClockwise)`: This draws a circle (or part of a circle). The circle starts from the current position of the path. Here, `aX` and `aY` are used as offsets from the current position. Note that `aRadius` sets the size of the circle and `aStartAngle` and `aEndAngle` define how large a part of the circle is drawn. The Boolean `aClockwise` property determines whether the circle is drawn clockwise or counterclockwise.
- `absArc(aX, aY, aRadius, aStartAngle, aEndAngle, AClockwise)`: See the description of the `arc` property. The position is absolute instead of relative to the current position.
- `ellipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)`: See the description of the `arc` property. As an addition, with the `ellipse` function, we can separately set the `x` radius and the `y` radius.
- `absEllipse(aX, aY, xRadius, yRadius, aStartAngle, aEndAngle, aClockwise)`: See the description of the `ellipse` property. The position is absolute instead of relative to the current position.
- `fromPoints(vectors)`: If you pass in an array of `THREE.Vector2` (or `THREE.Vector3`) objects into this function, Three.js will create a path using straight lines from the supplied vectors.

- `holes`: The `holes` property contains an array of `THREE.Shape` objects. Each of the objects in this array is rendered as a hole. A good example of this is the example we saw at the beginning of this section. In that code fragment, we added three `THREE.Shape` objects to this array: one for the left eye, one for the right eye, and one for the mouth of our main `THREE.Shape` object.

Just like for a lot of examples, to understand how the various properties affect the final shape, the easiest way is to just enable the `wireframe` property on the material and play around with the settings. For example, the following screenshot shows what happens when you use a low value for `curveSegments`:


localhost:8081/chapter-5/shape-ge
localhost:8081/chapter-5/shape-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
THREE.ShapeGeometry

emissive 000000
emissiveIntensity 1.635
roughness 0.078
metalness 1
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
curveSegments 2

Figure 5.8 – The wireframe of a shape geometry

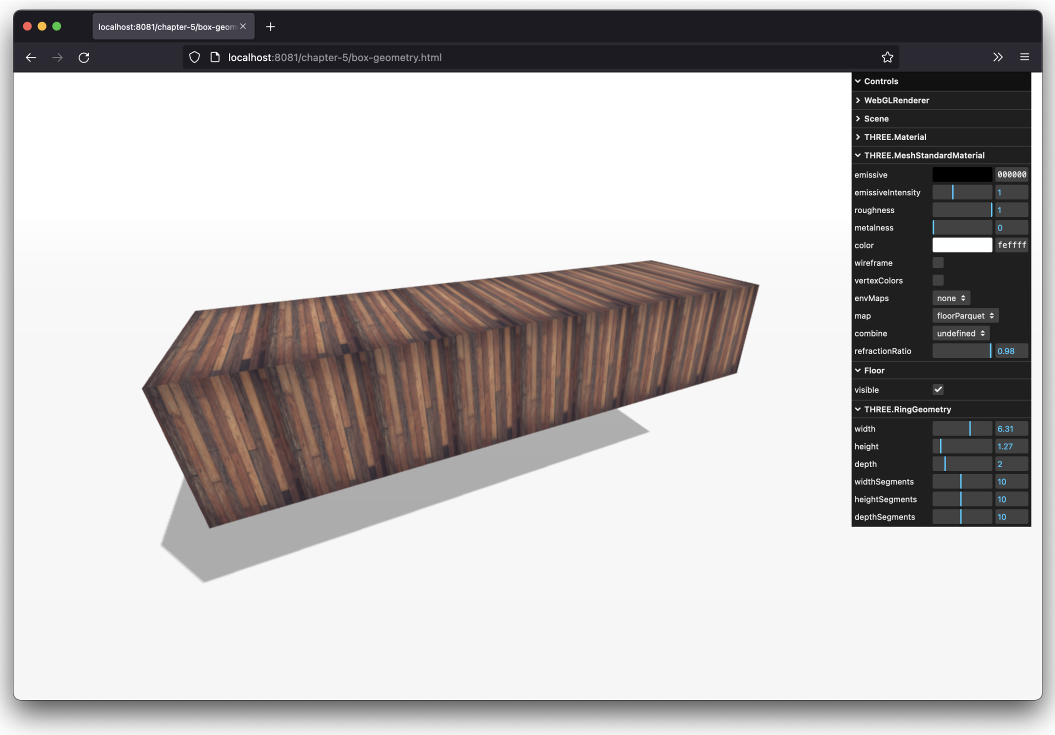
As you can see, the shape loses its nice, round edges, but uses a lot fewer faces in the process. That's it for the 2D shapes. The following sections will show and explain the basic 3D shapes.

3D geometries

In this section on basic 3D geometries, we'll start with a geometry we've already seen a couple of times: `THREE.BoxGeometry`.

THREE.BoxGeometry

`THREE.BoxGeometry` is a very simple 3D geometry that allows you to create a box by specifying its `width`, `height`, and `depth` properties. We've added an example, `box-geometry.html`, where you can play around with these properties. The following screenshot shows this geometry:


localhost:8081/chapter-5/box-geom
localhost:8081/chapter-5/box-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.RingGeometry

emissive: 000000
emissiveIntensity: 1
roughness: 1
metalness: 0
color: feffff
wireframe: ☐
vertexColors: ☐
envMaps: none
map: floorParquet
combine: undefined
refractionRatio: 0.98
visible: ☑
width: 6.31
height: 1.27
depth: 2
widthSegments: 10
heightSegments: 10
depthSegments: 10

Figure 5.9 – Basic 3D box geometry

As you can see in this example, by changing the `width`, `height`, and `depth` properties of `THREE.BoxGeometry`, you can control the size of the resulting mesh. These three properties are also mandatory when you create a new cube, as follows:

```
new THREE.BoxGeometry(10,10,10);
```

In the example, you can also see a couple of other properties that you can define on the cube. The following list explains all the properties:

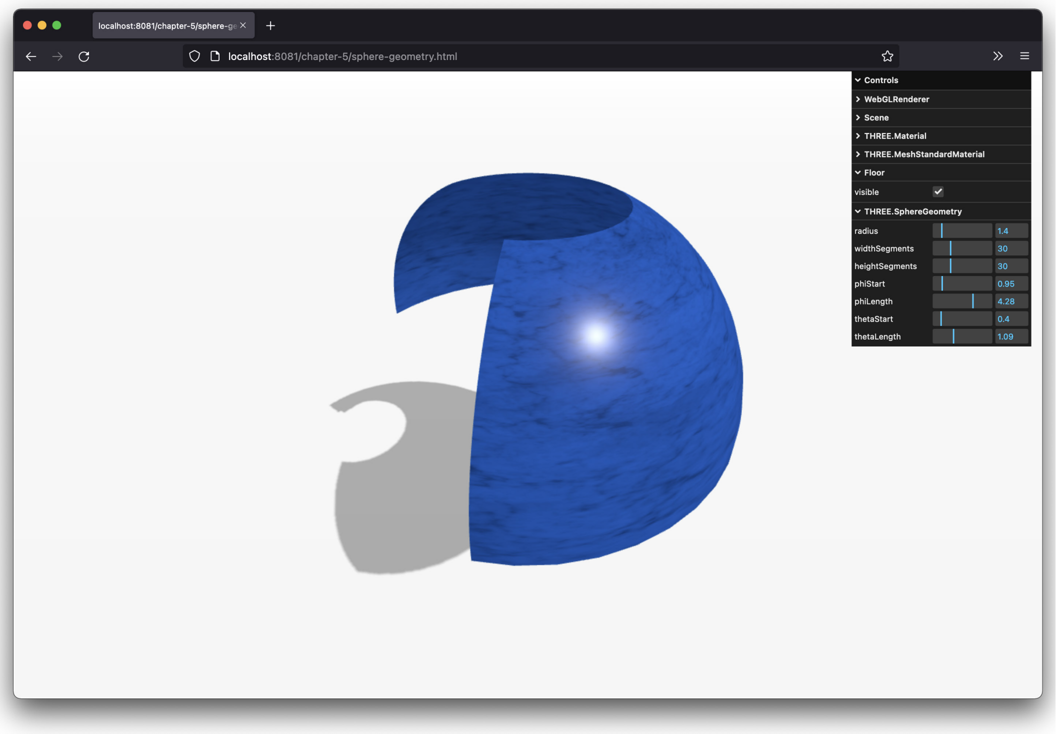
- `width`: This is the width of the cube. This is the length of the vertices of the cube along the *x*-axis.
- `height`: This is the height of the cube. This is the length of the vertices of the cube along the *y*-axis.
- `depth`: This is the depth of the cube. This is the length of the vertices of the cube along the *z*-axis.
- `widthSegments`: This is the number of segments into which we divide a face along the cube's *x*-axis. The default value is 1. The more segments you define, the more faces a side has. If this property and the next two are set to `1`, each side of the cube will just have 2 faces. If this property is set to `2`, the face will be divided into 2 segments, resulting in 4 faces.
- `heightSegments`: This is the number of segments into which we divide a face along the cube's *y*-axis. The default value is 1.
- `depthSegments`: This is the number of segments into which we divide a face along the cube's *z*-axis. The default value is 1.

By increasing the various segment properties, you divide the six main faces of the cube into smaller faces. This is useful if you want to set specific material properties on parts of the cube using `THREE.MeshFaceMaterial`.

`THREE.BoxGeometry` is a very simple geometry. Another simple one is `THREE.SphereGeometry`.

THREE.SphereGeometry

With `THREE.SphereGeometry`, you can create a 3D sphere. Let's dive straight into the example, `sphere-geometry.html`:


localhost:8081/chapter-5/sphere-ge
localhost:8081/chapter-5/sphere-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.SphereGeometry

visible ☑

radius 1.4

widthSegments 30

heightSegments 30

phiStart 0.95

phiLength 4.28

thetaStart 0.4

thetaLength 1.09

Figure 5.10 – Simple 3D sphere geometry

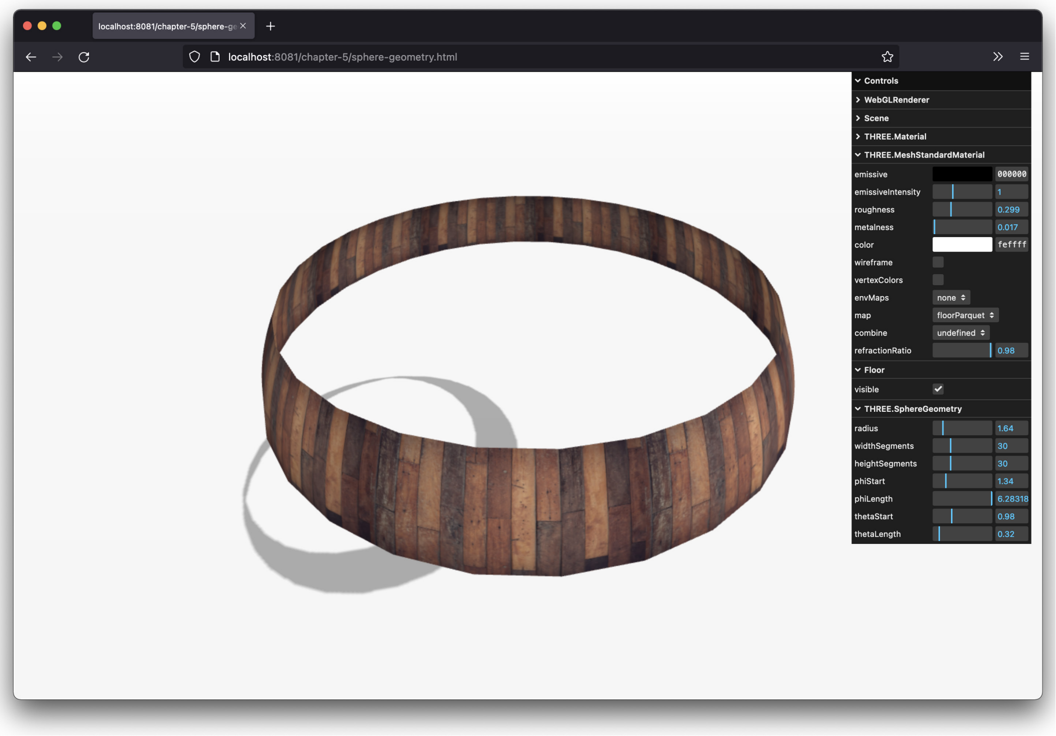
In the previous screenshot, we have shown you a half-open sphere that was created based on `THREE.SphereGeometry`. This geometry is a very flexible one and can be used to create all kinds of sphere-related geometries. A basic `THREE.SphereGeometry` instance, though, can be created as easily as this: `new THREE.SphereGeometry()`.

The following properties can be used to tune what the resulting mesh looks like:

- `radius`: This is used to set the radius for the sphere. This defines how large the resulting mesh will be. The default value is 50.
- `widthSegments`: This is the number of segments to be used vertically. More segments mean a smoother surface. The default value is 8 and the minimum value is 3.
- `heightSegments`: This is the number of segments to be used horizontally. The more segments, the smoother the surface of the sphere. The default value is 6 and the minimum value is 2.
- `phiStart`: This determines where to start drawing the sphere along its *x*-axis. This can range from 0 to `2 * PI`. The default value is `0`.

- `phiLength`: This determines how far from `phiStart` the sphere is to be drawn. `2 * PI` will draw a full sphere and `0.5 * PI` will draw an open quarter-sphere. The default value is `2 * PI`.
- `thetaStart`: This determines where to start drawing the sphere along its *x*-axis. This can range from 0 to `2 * PI`, and the default value is 0.
- `thetaLength`: This determines how far from `thetaStart` the sphere is drawn. The `2 * PI` value is a full sphere, whereas `PI` will draw only half of the sphere. The default value is `2 * PI`.

The `radius`, `widthSegments`, and `heightSegments` properties should be clear. We've already seen these kinds of properties in other examples. The `phiStart`, `phiLength`, `thetaStart`, and `thetaLength` properties are a bit harder to understand without looking at an example. Luckily, though, you can experiment with these properties from the menu in the `sphere-geometry.html` example and create interesting geometries such as these:


localhost:8081/chapter-5/sphere-ge
localhost:8081/chapter-5/sphere-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.SphereGeometry

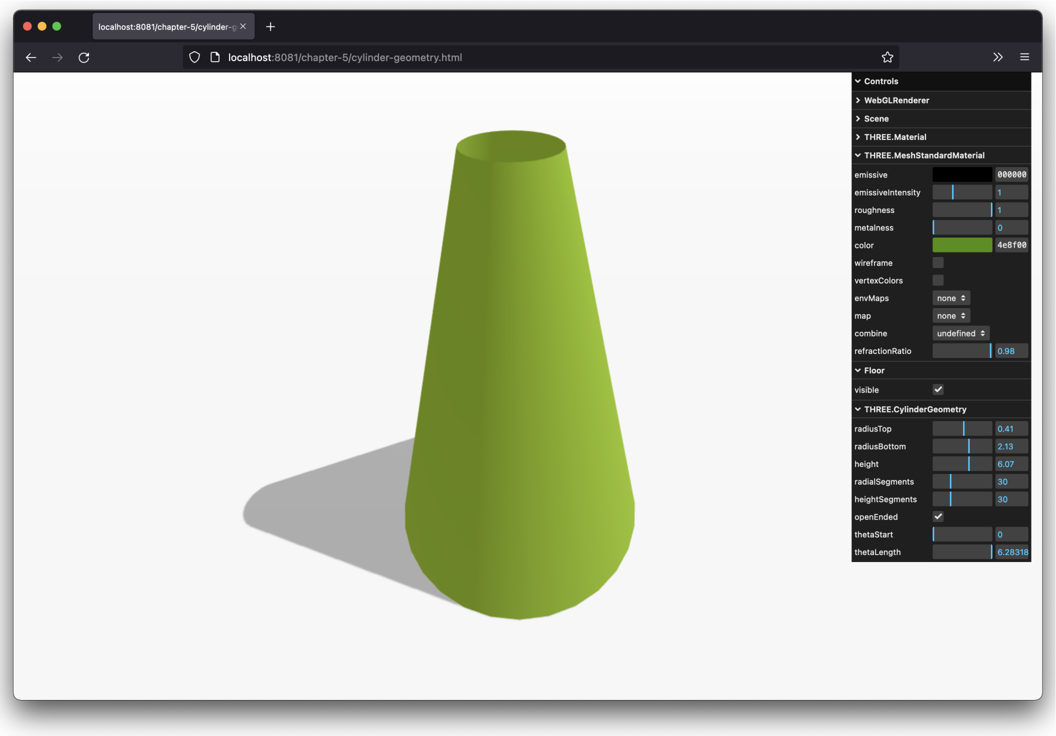
emissive 000000
emissiveIntensity 1
roughness 0.299
metalness 0.017
color feffff
wireframe ☐
vertexColors ☐
envMaps none
map floorParquet
combine undefined
refractionRatio 0.98
visible ☑
radius 1.64
widthSegments 30
heightSegments 30
phiStart 1.34
phiLength 6.28318
thetaStart 0.98
thetaLength 0.32

Figure 5.11 – Sphere geometry with different phi and theta properties

The next one on the list is `THREE.CylinderGeometry`.

THREE.CylinderGeometry

With this geometry, we can create cylinders and cylindrical objects. As for all the other geometries, we also have an example (`cylinder-geometry.html`) that lets you experiment with the properties of this geometry, the screenshot for which is as follows:


localhost:8081/chapter-5/cylinder-g
localhost:8081/chapter-5/cylinder-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.CylinderGeometry

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 4e8f00
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
radiusTop 0.41
radiusBottom 2.13
height 6.07
radialSegments 30
heightSegments 30
openEnded ☑
thetaStart 0
thetaLength 6.28318

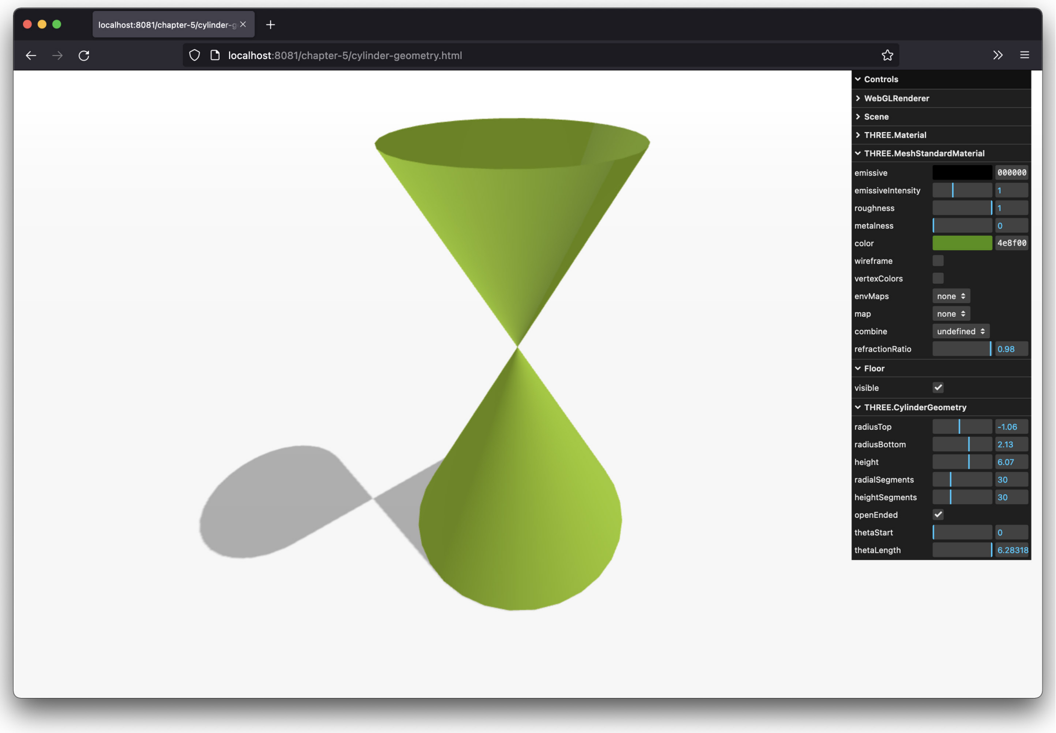
Figure 5.12 – 3D cylinder geometry

When you create `THREE.CylinderGeometry`, there aren't any mandatory arguments, so you can create a cylinder by just calling `new THREE.CylinderGeometry()`. You can pass in a number of properties, as you can see in the preceding example, to alter the appearance of this cylinder. The properties are explained in the following list:

- `radiusTop`: This sets the size that this cylinder will be at the top. The default value is 20.
- `radiusBottom`: This sets the size that this cylinder will be at the bottom. The default value is 20.
- `height`: This property sets the height of the cylinder. The default height is 100.
- `radialSegments`: This determines the number of segments along the radius of the cylinder. This defaults to 8. More segments mean a smoother cylinder.

- `heightSegments`: This determines the number of segments along the height of the cylinder. The default value is 1. More segments mean more faces.
- `openEnded`: This determines whether or not the mesh is closed at the top and the bottom. The default value is `false`.
- `thetaStart`: This determines where to start drawing the cylinder along its *x*-axis. This can range from 0 to `2 * PI`, and the default value is 0.
- `thetaLength`: This determines how far from `thetaStart` the cylinder is drawn. The `2 * PI` value is a full cylinder, whereas `PI` will draw only half of a cylinder. The default value is `2 * PI`.

These are all very basic properties that you can use to configure a cylinder. One interesting aspect, though, is when you use a negative `radius` value for the top (or for the bottom). If you do this, you can use this geometry to create an hourglass-like shape, as shown in the following screenshot:


localhost:8081/chapter-5/cylinder-g
localhost:8081/chapter-5/cylinder-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 4e8f00
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98

Floor

visible ☑

THREE.CylinderGeometry

radiusTop -1.06
radiusBottom 2.13
height 6.07
radialSegments 30
heightSegments 30
openEnded ☑
thetaStart 0
thetaLength 6.28318

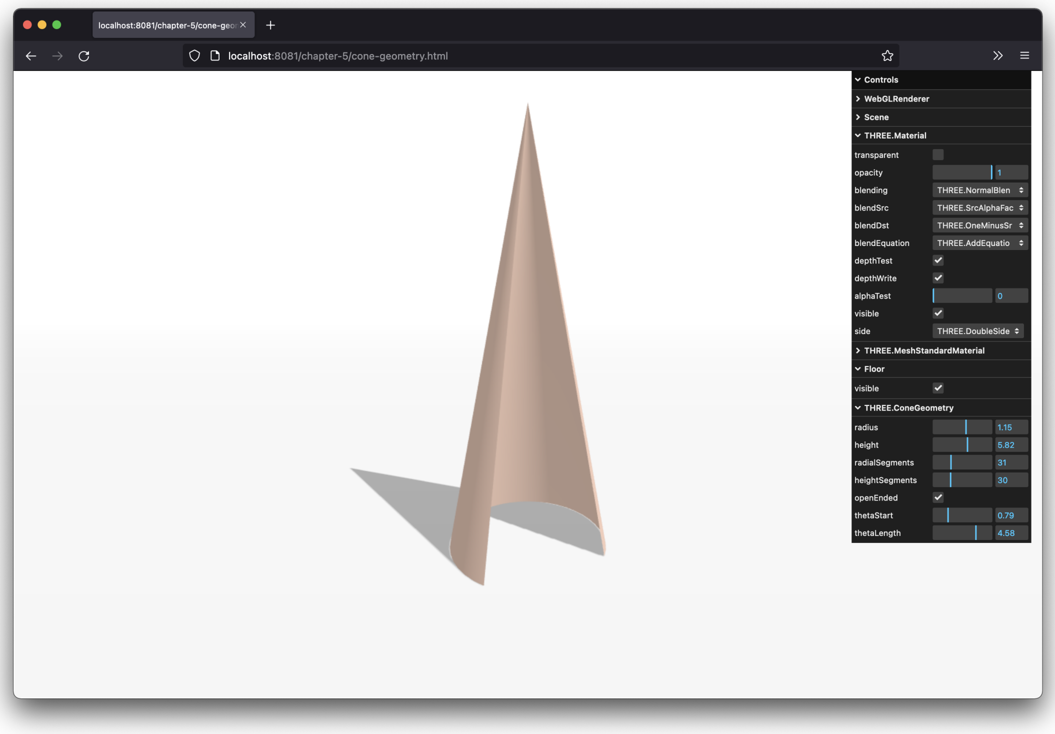
Figure 5.13 – 3D cylinder geometry as an hourglass

One thing to note here is that the top half, in this case, is turned inside out. If you use a material that isn't configured with `THREE.DoubleSide`, you won't see the top half.

The next geometry is `THREE.ConeGeometry`, which provides the basic functionalities of `THREE.CylinderGeometry`, but has the top radius fixed to zero.

THREE.ConeGeometry

`THREE.ConeGeometry` is pretty much the same as `THREE.CylinderGeometry`. It uses all the same properties, except it only allows you to set the radius instead of separate `radiusTop` and `radiusBottom` values:


localhost:8081/chapter-5/cone-geo
localhost:8081/chapter-5/cone-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.ConeGeometry

transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.DoubleSide

visible ☑

radius 1.15
height 5.82
radialSegments 31
heightSegments 30
openEnded ☑
thetaStart 0.79
thetaLength 4.58

Figure 5.14 – Simple 3D cone geometry

The following properties can be set on `THREE.ConeGeometry`:

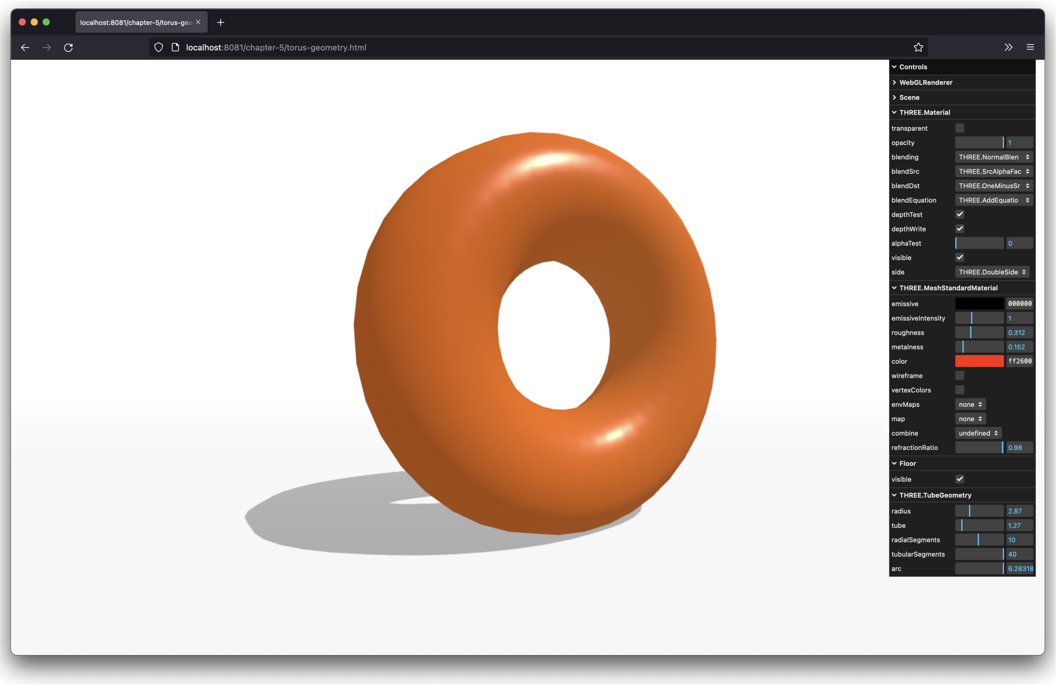
- `radius`: This sets the size that this cylinder will be at the bottom. The default value is `20`.
- `height`: This property sets the height of the cylinder. The default height is 100.
- `radialSegments`: This determines the number of segments along the radius of the cylinder. This defaults to 8. More segments mean a smoother cylinder.

- `heightSegments`: This determines the number of segments along the height of the cylinder. The default value is 1. More segments mean more faces.
- `openEnded`: This determines whether or not the mesh is closed at the top and the bottom. The default value is `false`.
- `thetaStart`: This determines where to start drawing the cylinder along its *x*-axis. This can range from 0 to `2 * PI`, and the default value is 0.
- `thetaLength`: This determines how far from `thetaStart` the cylinder is drawn. The `2 * PI` value is a full cylinder, whereas `PI` will draw only half of a cylinder. The default value is `2 * PI`.

The next geometry, `THREE.TorusGeometry`, allows you to create a donut-like shaped object.

THREE.TorusGeometry

A torus is a simple shape that looks like a donut. The following screenshot, which you can get yourself by opening the `torus-geometry.html` example, shows `THREE.TorusGeometry` in action:


localhost:8081/chapter-5/torus-geo
localhost:8081/chapter-5/torus-geometry.html

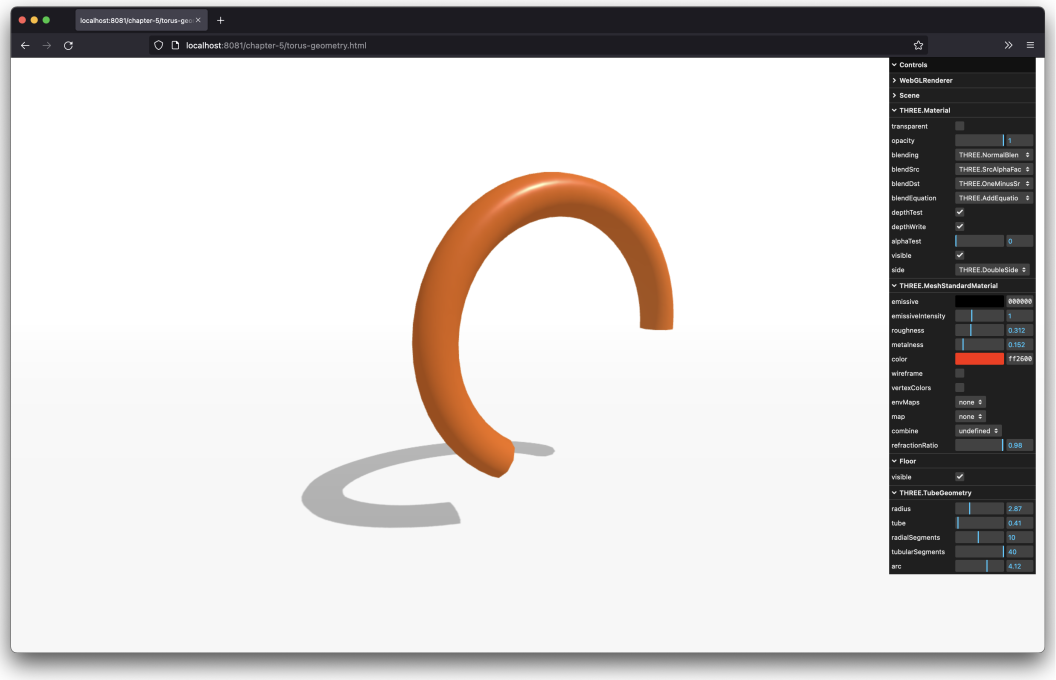
Controls	
WebGLRenderer	
Scene	
THREE.Material	
transparent	☐
opacity	1
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.DoubleSide
THREE.MeshStandardMaterial	
emissive	000000
emissiveIntensity	1
roughness	0.312
metalness	0.152
color	ff2600
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	undefined
refractionRatio	0.98
Floor	
visible	☑
THREE.TubeGeometry	
radius	2.87
tube	1.27
radialSegments	10
tubularSegments	40
arc	6.28318

Figure 5.15 – 3D torus geometry

Just like most of the simple geometries, there aren't any mandatory arguments when creating `THREE.TorusGeometry`. The following list mentions the arguments you can specify when you create this geometry:

- `radius`: This sets the size of the complete torus. The default value is 100.
- `tube`: This sets the radius of the tube (the actual donut). The default value for this attribute is 40.
- `radialSegments`: This determines the number of segments to be used along the length of the torus. The default value is 8. See the effect of changing this value in the example.
- `tubularSegments`: This determines the number of segments to be used along the width of the torus. The default value is 6. See the effect of changing this value in the example.
- `arc`: With this property, you can control whether the torus draws a full circle. The default of this value is `2 * PI` (a full circle).

Most of these are very basic properties that you've already seen. The `arc` property, however, is a very interesting one. With this property, you can define whether the donut makes a full circle or only a partial one. By experimenting with this property, you can create very interesting meshes, such as the following one with `arc` set to a value lower than `2 * PI`:


localhost:8081/chapter-5/torus-ge
localhost:8081/chapter-5/torus-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.TubeGeometry

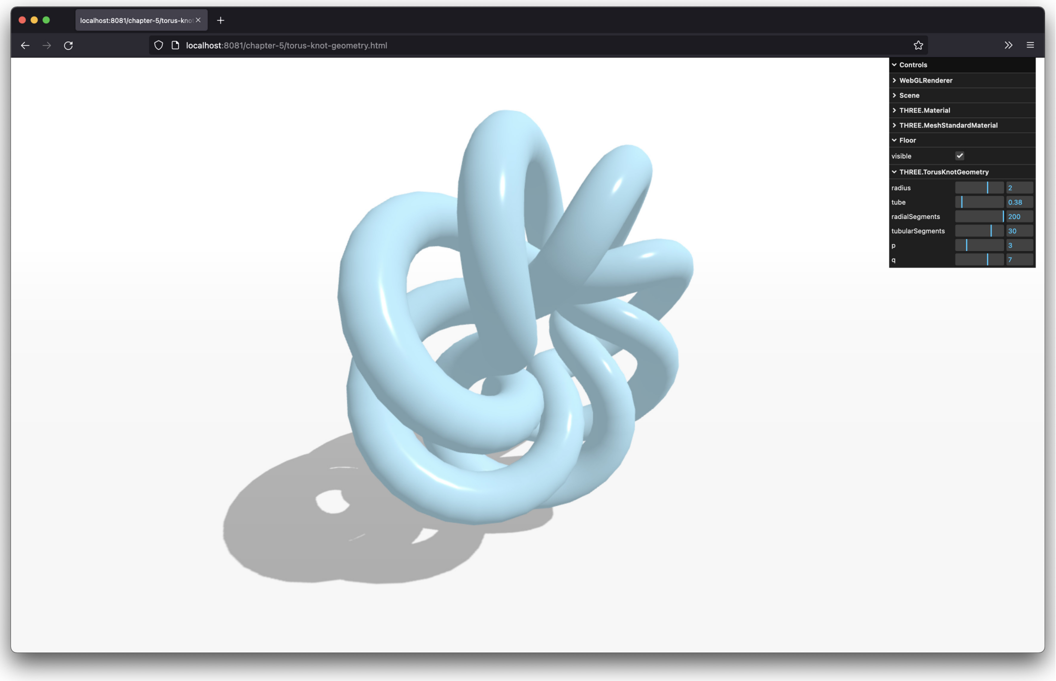
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.DoubleSide
emissive 000000
emissiveIntensity 1
roughness 0.312
metalness 0.152
color ff2600
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
radius 2.87
tube 0.41
radialSegments 10
tubularSegments 40
arc 4.12

Figure 5.16 – 3D torus geometry with a smaller arc property

`THREE.TorusGeometry` is a very straightforward geometry. In the next section, we'll look at a geometry that almost shares its name but is much less straightforward: `THREE.TorusKnotGeometry`.

THREE.TorusKnotGeometry

With `THREE.TorusKnotGeometry`, you can create a torus knot. A torus knot is a special kind of knot that looks like a tube that winds around itself a couple of times. The best way to explain this is by looking at the `torus-knot-geometry.html` example. The following screenshot shows this geometry:

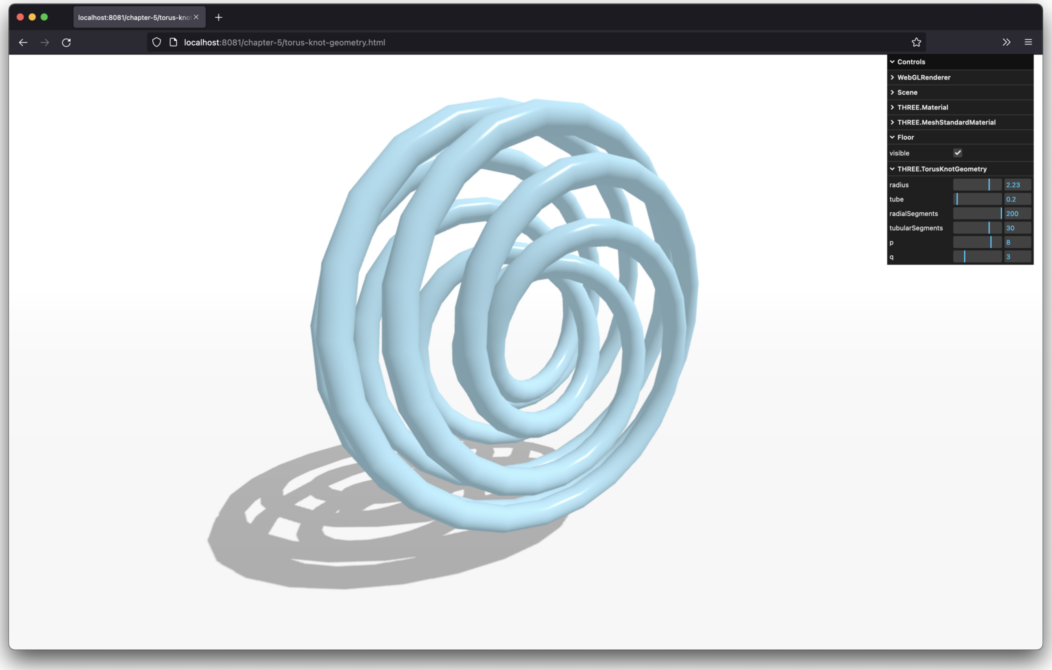

localhost:8081/chapter-5/torus-knot-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.TorusKnotGeometry

visible ☑
radius 2
tube 0.38
radialSegments 200
tubularSegments 30
p 3
q 7

Figure 5.17 – Torus knot geometry

If you open this example and play around with the `p` and `q` properties, you can create all kinds of beautiful geometries. The `p` property defines how often the knot winds around its axis, and `q` defines how much the knot winds around its interior.

If this sounds a bit vague, don't worry. You don't need to understand these properties to create beautiful knots, such as the one shown in the following screenshot (for those interested in the details, Wolfram has a good article on this subject at `https://mathworld.wolfram.com/TorusKnot.html`):


localhost:8081/chapter-5/torus-knot
localhost:8081/chapter-5/torus-knot-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.TorusKnotGeometry

visible ☑

radius 2.23
tube 0.2
radialSegments 200
tubularSegments 30
p 8
q 3

Figure 5.18 – Torus knot geometry with different p and q values

With the example for this geometry, you can play around with the following properties and see the effect that various combinations of `p` and `q` have on this geometry:

- `radius`: This sets the size of the complete torus. The default value is 100.
- `tube`: This sets the radius of the tube (the actual donut). The default value for this attribute is 40.
- `radialSegments`: This determines the number of segments to be used along the length of the torus knot. The default value is 64. See the effect of changing this value in the demo.
- `tubularSegments`: This determines the number of segments to be used along the width of the torus knot. The default value is 8. See the effect of changing this value in the demo.
- `p`: This defines the shape of the knot, and the default value is 2.
- `q`: This defines the shape of the knot, and the default value is 3.
- `heightScale`: With this property, you can stretch out the torus knot. The default value is 1.

The next geometry on the list is the last one of the basic geometries: `THREE.PolyhedronGeometry`.

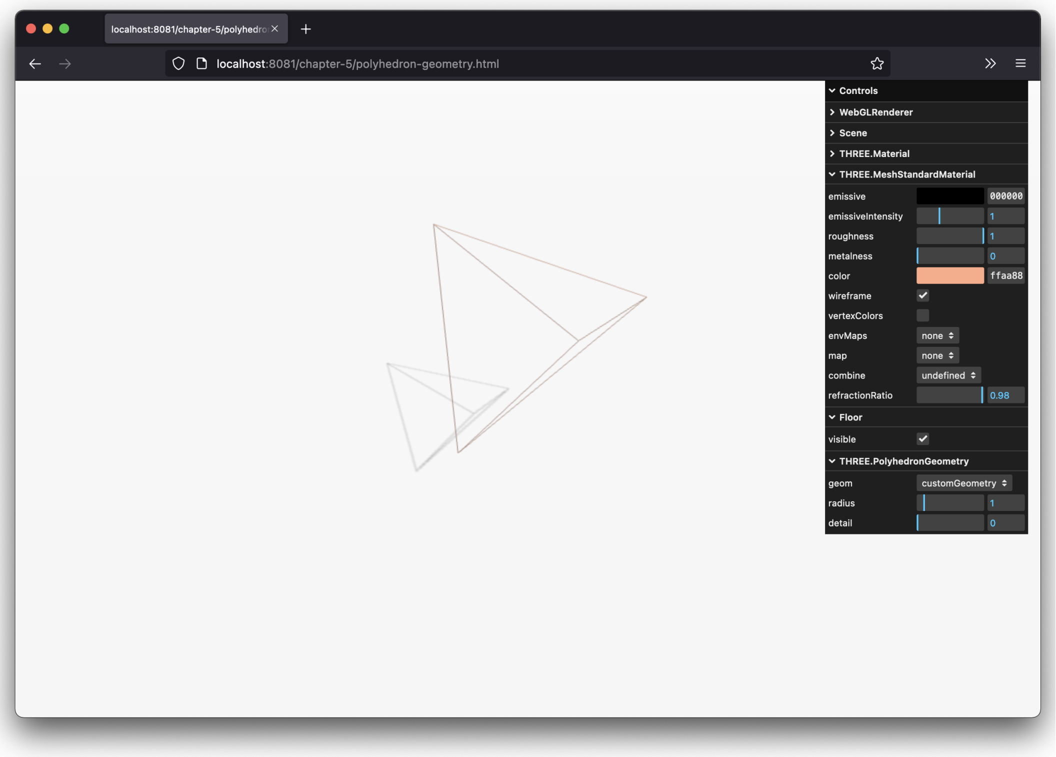
THREE.PolyhedronGeometry

With this geometry, you can easily create polyhedrons. A polyhedron is a geometry that has only flat faces and straight edges. Most often, though, you won't use `THREE.PolyhedronGeometry` directly. Three.js provides a number of specific polyhedrons you can use without having to specify the vertices and faces of `THREE.PolyhedronGeometry`. We'll discuss these polyhedrons later on in this section.

If you do want to use `THREE.PolyhedronGeometry` directly, you have to specify the vertices and the faces (just as we did for the cube in *Chapter 3*, *Working with Light Sources in Three.js*). For instance, we can create a simple tetrahedron (also see the section on `THREE.TetrahedronGeometry` in this chapter) like this:

```
const vertices = [
 1,  1,  1,
-1, -1,  1,
-1,  1, -1,
 1, -1, -1
];
const indices = [
2,  1,  0,
0,  3,  2,
1,  3,  0,
2,  3,  1
];
new THREE.PolyhedronBufferGeometry(vertices, indices, radius,
detail)
```

To construct `THREE.PolyhedronGeometry`, we pass in the `vertices`, `indices`, `radius`, and `detail` properties. The resulting `THREE.PolyhedronGeometry` object is shown in the `polyhedron-geometry.html` example:



localhost:8081/chapter-5/polyhedron-geometry.html

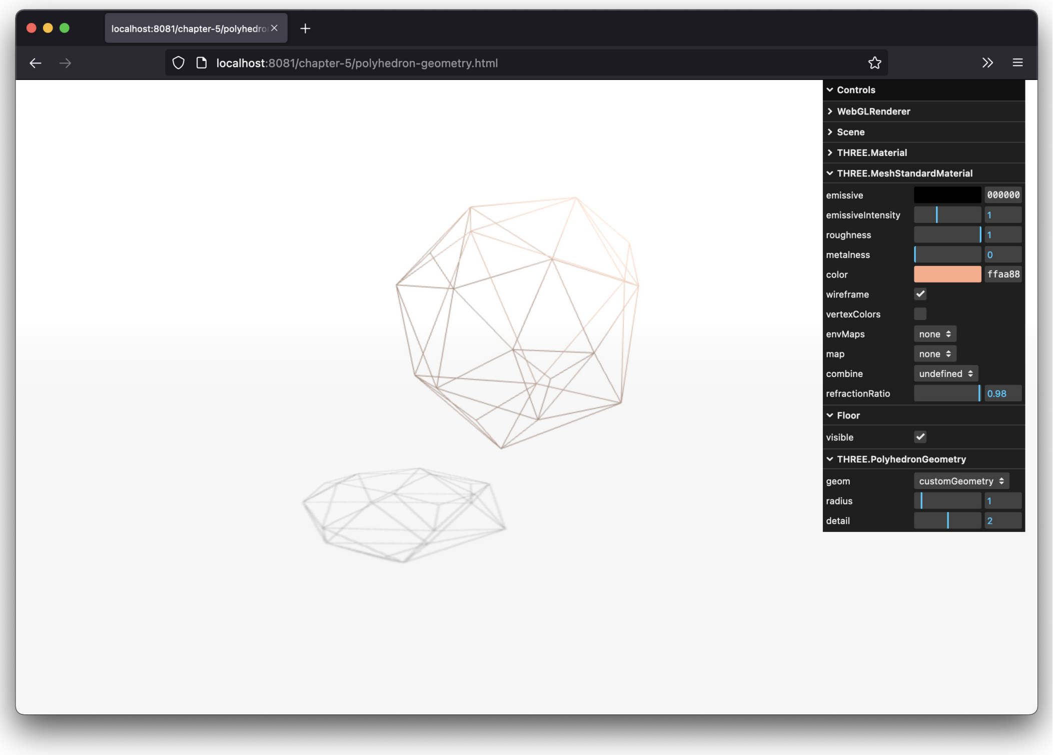
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color ffaa88
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
Floor
visible ☑
THREE.PolyhedronGeometry
geom customGeometry
radius 1
detail 0

Figure 5.19 – Custom polyhedron

When you create a polyhedron, you can pass in the following four properties:

- `vertices`: These are the points that make up the polyhedron.
- `indices`: These are the faces that need to be created from the vertices.
- `radius`: This is the size of the polyhedron. This defaults to 1.
- `detail`: With this property, you can add additional detail to the polyhedron. If you set this to `1`, each triangle in the polyhedron will be split into 4 smaller triangles. If you set this to `2`, those 4 smaller triangles will each be split into 4 smaller triangles once more, and so on.

The following screenshot shows the same custom mesh, but now with a higher level of detail:


localhost:8081/chapter-5/polyhedro
localhost:8081/chapter-5/polyhedron-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PolyhedronGeometry

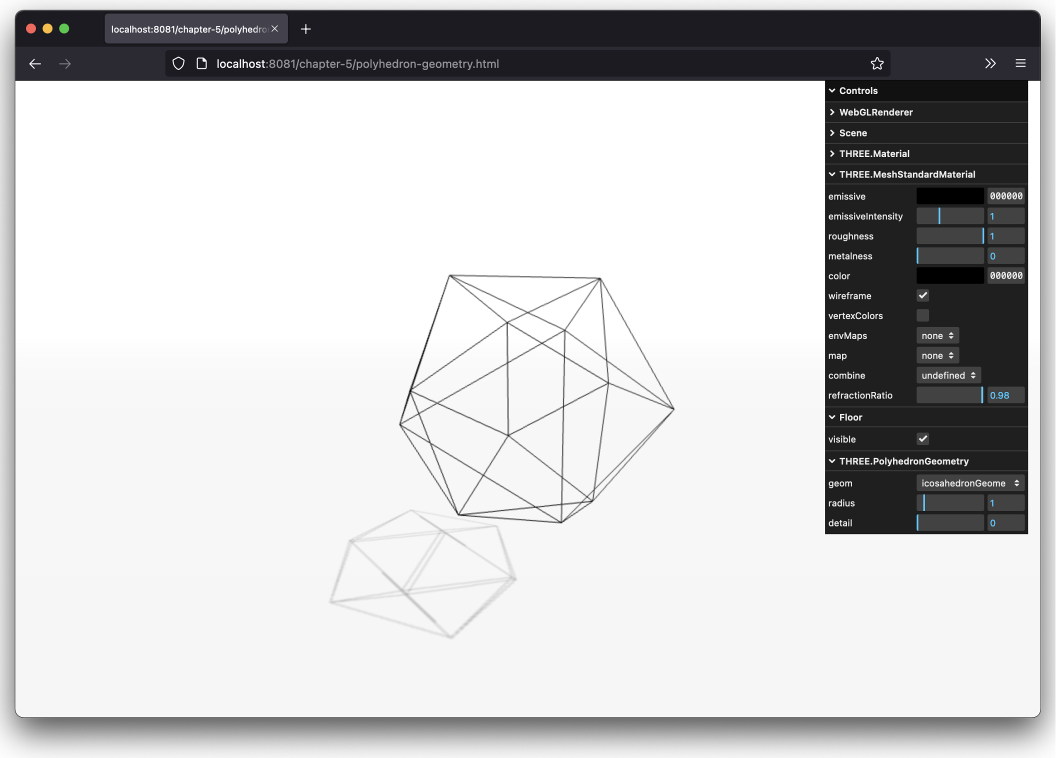
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color ffaa88
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
geom customGeometry
radius 1
detail 2

Figure 5.20 – A custom polyhedron with a higher level of detai

At the beginning of this section, we mentioned that Three.js comes with a couple of polyhedrons out of the box. In the following subsections, we'll quickly show you these. All of these polyhedron types can be viewed by looking at the `polyhedron-geometry.html` example.

THREE.IcosahedronGeometry

`THREE.IcosahedronGeometry` creates a polyhedron that has 20 identical triangular faces created from 12 vertices. When creating this polyhedron, all you need to specify are the radius and the detail level. This screenshot shows a polyhedron that was created using `THREE.IcosahedronGeometry`:


localhost:8081/chapter-5/polyhedro
localhost:8081/chapter-5/polyhedron-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PolyhedronGeometry

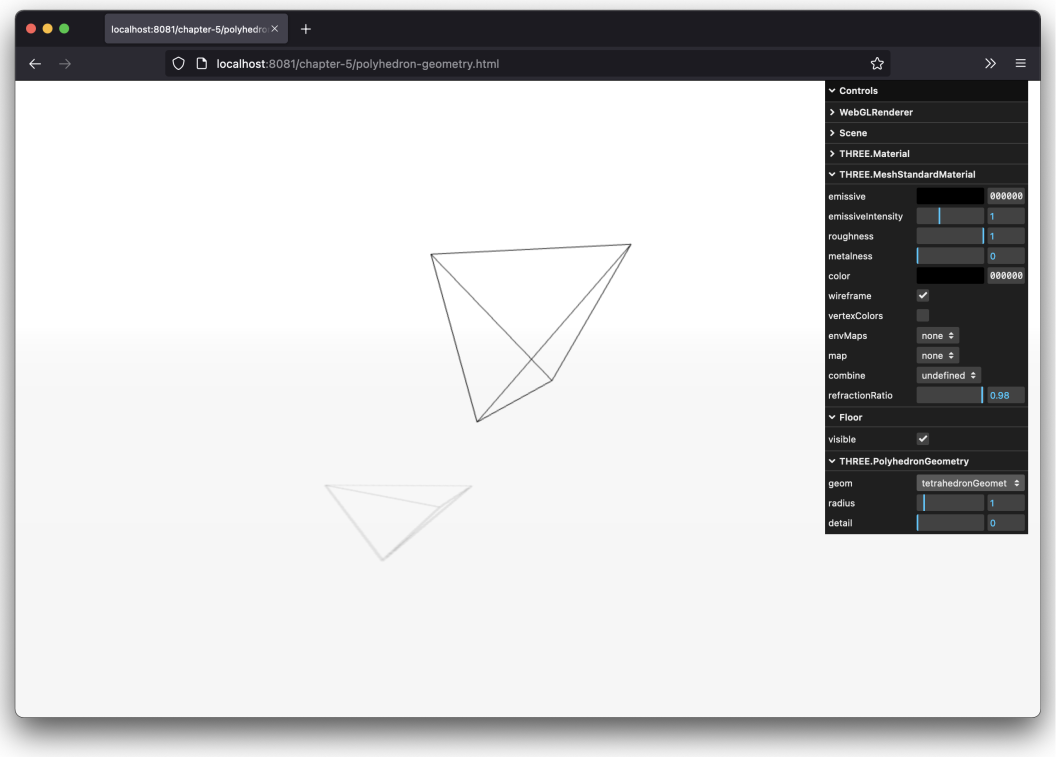
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
geom icosahedronGeome
radius 1
detail 0

Figure 5.21 – 3D icosahedron geometry

The next polyhedron we're going to look at is `THREE.TetrahedronGeometry`.

THREE.TetrahedronGeometry

The tetrahedron is one of the simplest polyhedrons. This polyhedron only contains four triangular faces that are created from four vertices. You can create `THREE.TetrahedronGeometry` just like the other polyhedrons provided by Three.js, by specifying the radius and detail levels. Here's a screenshot that shows a tetrahedron that was created using `THREE.TetrahedronGeometry`:


localhost:8081/chapter-5/polyhedro
localhost:8081/chapter-5/polyhedron-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PolyhedronGeometry

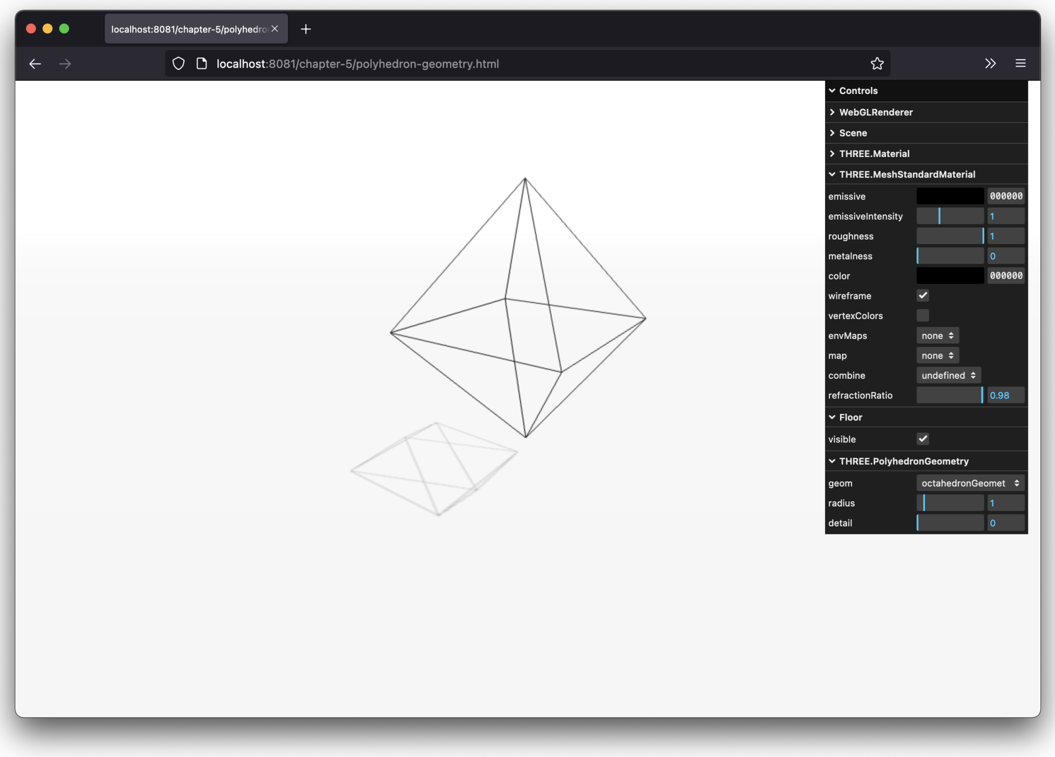
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
geom tetrahedronGeomet
radius 1
detail 0

Figure 5.22 – 3D tetrahedron geometry

Next, we will see a polyhedron with eight faces: an octahedron.

THREE.OctahedronGeometry

Three.js also provides an implementation of an octahedron. As the name implies, this polyhedron has eight faces. These faces are created from six vertices. The following screenshot shows this geometry:


localhost:8081/chapter-5/polyhedron-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.PolyhedronGeometry

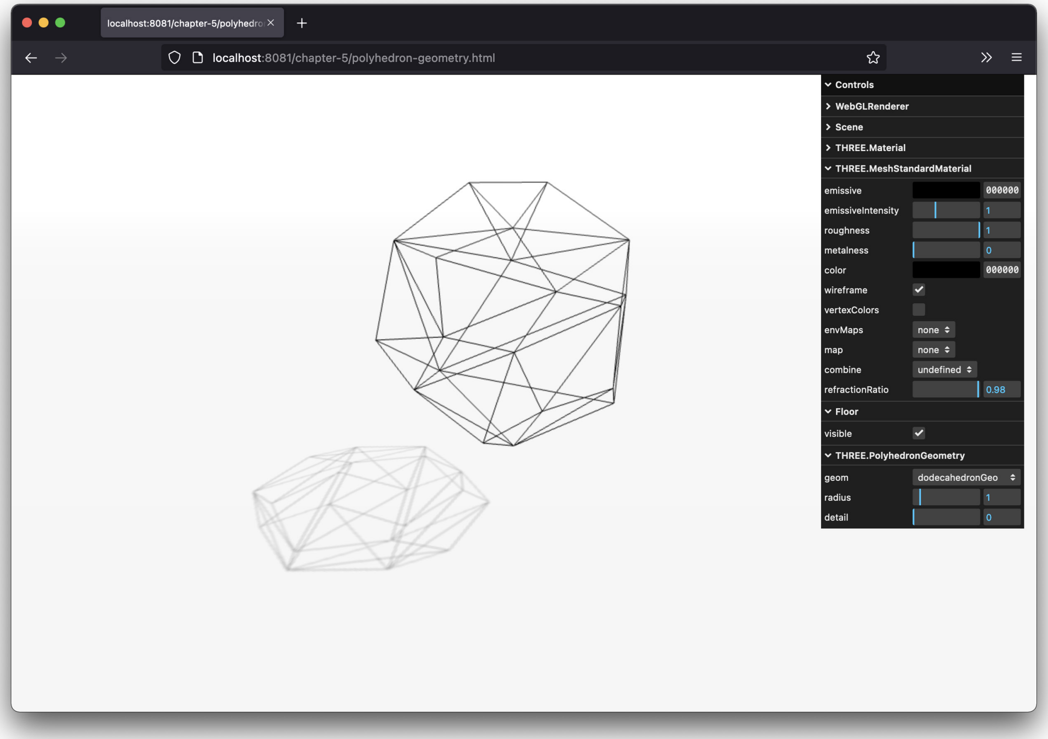
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
geom octahedronGeomet
radius 1
detail 0

Figure 5.23 – 3D octahedron geometry

The last of the polyhedrons we're going to look at is a dodecahedron.

THREE.DodecahedronGeometry

The final polyhedron geometry provided by Three.js is `THREE.DodecahedronGeometry`. This polyhedron has 12 faces. The following screenshot shows this geometry:


localhost:8081/chapter-5/polyhedro
localhost:8081/chapter-5/polyhedron-geometry.html

Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
Floor
visible ☑
THREE.PolyhedronGeometry
geom dodecahedronGeo
radius 1
detail 0

Figure 5.24 – 3D dodecahedron geometry

As you can see, Three.js provides a large number of 3D geometries, from directly useful ones such as spheres and cubes to geometries that probably aren't that useful in practice, such as the set of polyhedrons. In any case, these 3D geometries will give you a good starting point for creating and experimenting with materials, geometries, and 3D scenes.

Summary

In this chapter, we discussed all of the standard geometries that Three.js has to offer. As you saw, there are a whole lot of geometries you can use right out of the box. To best learn how to use the geometries, experiment with them. Use the examples in this chapter to get to know the properties you can use to customize the standard set of geometries available from Three.js.

For 2D shapes, it's important to remember that they are placed on the *x-y* plane. If you want to have a 2D shape horizontally, you'll have to rotate the mesh around the *x*-axis for `-0.5 * PI`. And finally, take care that if you're rotating a 2D shape, or a 3D shape that is open (for example, a cylinder or a tube), remember to set the material to `THREE.DoubleSide`. If you don't do this, the inside or the back of your geometry won't be shown.

In this chapter, we focused on simple, straightforward meshes. Three.js also provides ways to create complex geometries, which we will cover in *Chapter 6*.

6

Exploring Advanced Geometries

In *Chapter 5*, *Learning to Work with Geometries*, we showed you all the basic geometries provided by Three.js. Besides these basic geometries, Three.js also offers a set of more advanced and specialized objects.

In this chapter, we'll show you these advanced geometries:

- How to use advanced geometries such as `THREE.ConvexGeometry`, `THREE.LatheGeometry`, `THREE.BoxLineGeometry`, `THREE.RoundeBoxGeometry`, `THREE.TeapotGeometry`, and `THREE.TubeGeometry`.
- How to create 3D shapes from 2D shapes using `THREE.ExtrudeGeometry`. We'll create a 3D shape from a 2D SVG image, and we will extrude from 2D Three.js shapes to create novel 3D shapes.
- If you want to create custom shapes yourself, you can continue playing with the ones we've discussed in the previous chapters. Three.js, however, also offers a `THREE.ParametricGeometry` object. With parametric geometry, you can create geometry with parameters you can change to effect the shape of the geometry.
- We'll also show how you can create 3D text effects using `THREE.TextGeometry`, and show you how to use the Troika library for when you want to add 2D text labels to your scene.
- Additionally, we'll show you how you can use two helper geometries, `THREE.WireframeGeometry` and `THREE.EdgesGeometry`. These helpers allow you to see more details about other geometries.

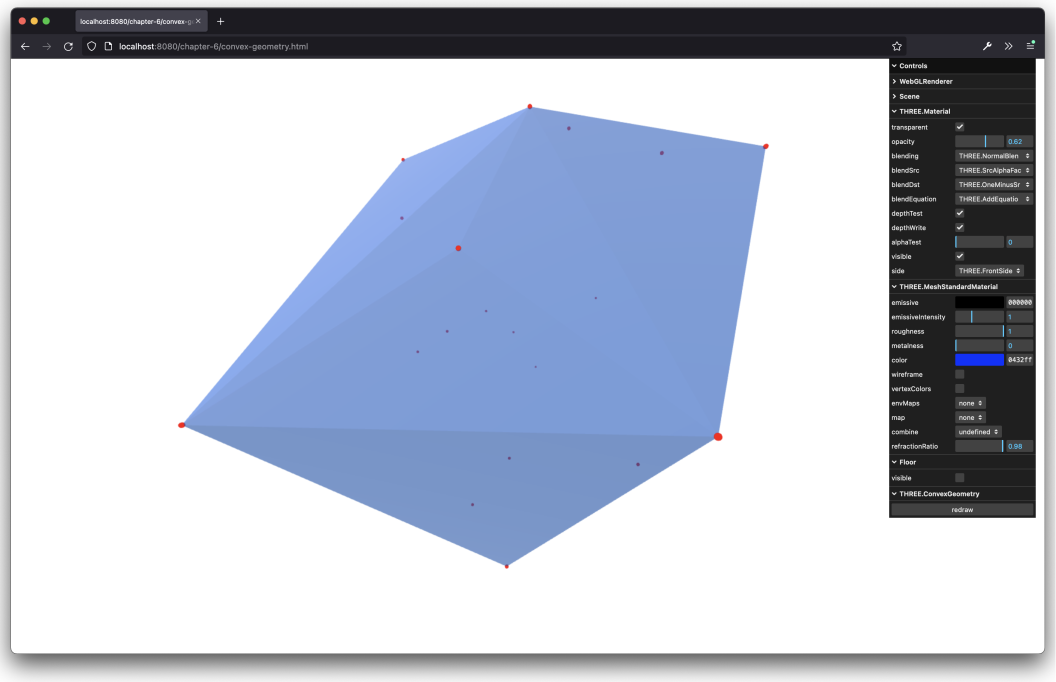
We'll start with the first one from this list, `THREE.ConvexGeometry`.

Learning advanced geometries

In this section, we'll look at a number of advanced Three.js geometries. We will start with `THREE.ConvexGeometry`, which you can use to create convex hulls.

THREE.ConvexGeometry

With `THREE.ConvexGeometry`, we can create a convex hull from a set of points. A convex hull is the minimal shape that encompasses all these points. The easiest way to understand this is by looking at an example. If you open up the `convex-geometry.html` example, you'll see the convex hull for a random set of points. The following screenshot shows this geometry:


localhost:8080/chapter-6/convex-g
localhost:8080/chapter-6/convex-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.ConvexGeometry
redraw

THREE.Material	
transparent	☑
opacity	0.62
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide

THREE.MeshStandardMaterial	
emissive	000000
emissiveIntensity	1
roughness	1
metalness	0
color	0432ff
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	undefined
refractionRatio	0.98

Floor	
visible	☐

Figure 6.1 – The convex hull encompassing all the points

In this example, we generate a random set of points, and based on these points, we create `THREE.ConvexGeometry`. In the example, you can use the **redraw** button in the menu on the right, which will generate 20 new points and draw the convex hull. If you try this for yourself, enable the material's transparency and set the opacity to a level below `1` to see the points that are used to create this geometry. These points are created as small `THREE.SphereGeometry` objects for this example.

To create `THREE.ConvexGeometry`, we need a set of points. The following code fragment shows how we do this:

```
const generatePoints = () => {
  const spGroup = new THREE.Object3D()
  spGroup.name = 'spGroup'
  const points = []
  for (let i = 0; i < 20; i++) {
    const randomX = -5 + Math.round(Math.random() * 10)
    const randomY = -5 + Math.round(Math.random() * 10)
    const randomZ = -5 + Math.round(Math.random() * 10)
    points.push(new THREE.Vector3(randomX, randomY, randomZ))
  }
  const material = new THREE.MeshBasicMaterial({ color:
    0xff0000, transparent: false })
  points.forEach(function (point) {
    const spGeom = new THREE.SphereGeometry(0.04)
    const spMesh = new THREE.Mesh(spGeom, material)
    spMesh.position.copy(point)
    spGroup.add(spMesh)
  })
  return {
    spGroup,
    points
  }
}
```

As you can see in this snippet of code, we create 20 random points (`THREE.Vector3`), which we push into an array. Next, we iterate this array and create `THREE.SphereGeometry`, whose position we set to one of these points (`position.copy(point)`). All the points are added to a group, so we can easily replace them once we do a redraw. Once you have this set of points, creating a `THREE.ConvexGeometry` from them is very easy, as shown in the following code snippet:

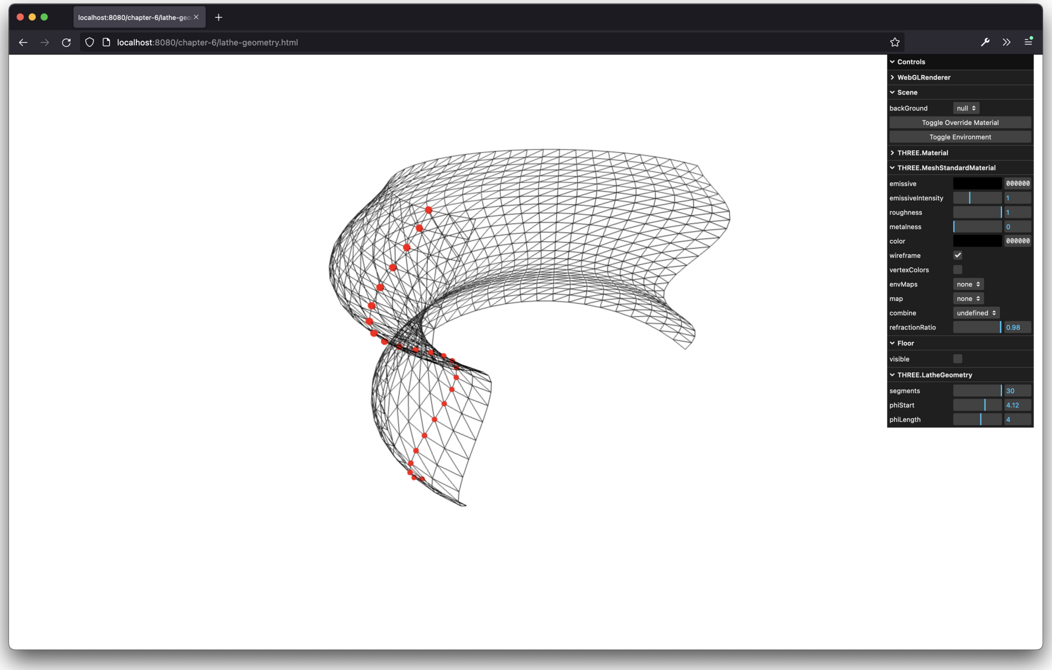
```
const convexGeometry = new THREE.ConvexGeometry(points);
```

An array containing vertices (of the `THREE.Vector3` type) is the only argument `THREE.ConvexGeometry` takes. Note that if you want to render a smooth `THREE.ConvexGeometry`, you should call `computeVertexNormals`, as we explained in *Chapter 2*, *The Basic Components that Make up a Three.js Application*.

The next complex geometry is `THREE.LatheGeometry`, which, for example, can be used to create vase-like shapes.

THREE.LatheGeometry

`THREE.LatheGeometry` allows you to create shapes from a set of points that together form a curve. If you look at *Figure 6.2*, you can see that we created a number of points (the red dots), which Three.js uses to create `THREE.LatheGeometry`. Once again, the easiest way to understand what `THREE.LatheGeometry` looks like is by looking at an example. This geometry is shown in `lathe-geometry.html`. The following screenshot taken from the example shows this geometry:


localhost:8080/chapter-6/lathe-geo
localhost:8080/chapter-6/lathe-geometry.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.LatheGeometry

backGround: null
emissive: 000000
emissiveIntensity: 1
roughness: 1
metalness: 0
color: 000000
wireframe: ☑
vertexColors: ☐
envMaps: none
map: none
combine: undefined
refractionRatio: 0.98
visible: ☐
segments: 30
phiStart: 4.12
phiLength: 4

Figure 6.2 – A lathe for vase-like meshes

In the preceding screenshot, you can see the points used to create this geometry as a set of small red spheres. The positions of these points are passed into `THREE.LatheGeometry`, together with arguments that define the shape of the geometry. Before we look at all the arguments, let's look at the code used to create the individual points and how `THREE.LatheGeometry` uses these points:

```
const generatePoints = () => {
  ...
  const points = []
  const height = 0.4
  const count = 25
  for (let i = 0; i < count; i++) {
    points.push(new THREE.Vector3((Math.sin(i * 0.4) +
      Math.cos(i * 0.4)) * height + 3, i / 6, 0))
  }
  ...
}
// use the same points to create a LatheGeometry
const latheGeometry = new THREE.LatheGeometry (points,
  segments, phiStart, phiLength);
latheMesh = createMesh(latheGeometry);
scene.add(latheMesh);
}
```

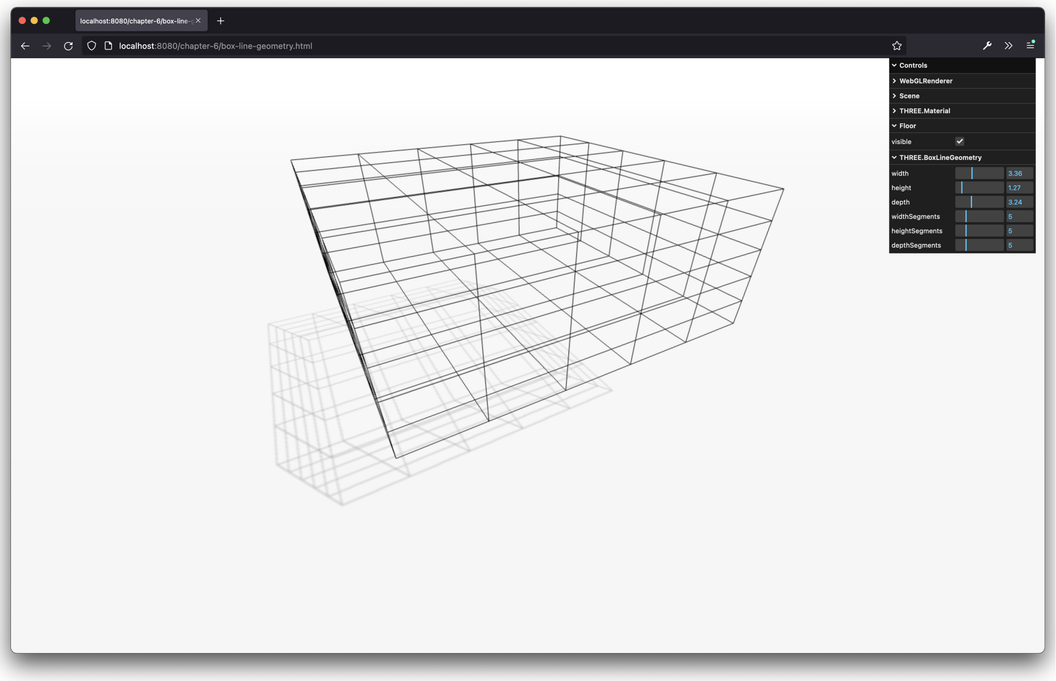
In this piece of JavaScript, we can see that we generate 25 points whose *x* coordinate is based on a combination of sine and cosine functions, while the *y* coordinate is based on the `i` and `count` variables. This creates a spline visualized by the red dots in the preceding screenshot. Based on these points, we can create `THREE.LatheGeometry`. Besides the array of vertices, `THREE.LatheGeometry` takes a couple of other arguments. The following list explains these properties:

- `points`: These are the points that make up the spline used to generate the bell/vase shape.
- `segments`: These are the number of segments used when creating the shape. The higher this number, the more round and smooth the resulting shape will be. The default value for this is `12`.
- `phiStart`: This determines where to start on a circle when generating the shape. This can range from `0` to `2*PI`. The default value is `0`.
- `phiLength`: This defines how fully generated the shape is. For instance, a quarter shape will be `0.5*PI`. The default value is the full 360 degrees or `2*PI`. This shape will start at the position of the `phiStart` property.

In *Chapter 5*, we've already seen `BoxGeometry`. Three.js also provides two other box-like geometries, which we'll discuss next.

BoxLineGeometry

If you just want to show the outline, you can use `THREE.BoxLineGeometry`. This geometry works exactly like `THREE.BoxGeometry`, but instead of rendering a solid object, it renders the box using lines like this (from `box-line-geometry.html`):


localhost:8080/chapter-6/box-line-
localhost:8080/chapter-6/box-line-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
Floor
THREE.BoxLineGeometry

visible ☑
width 3.36
height 1.27
depth 3.24
widthSegments 5
heightSegments 5
depthSegments 5

Figure 6.3 – A box rendered using lines

You use this geometry in the same way as `THREE.BoxGeometry`, but instead of creating `THREE.Mesh`, we need to create `THREE.LineSegments`, using one of the available line-specific materials:

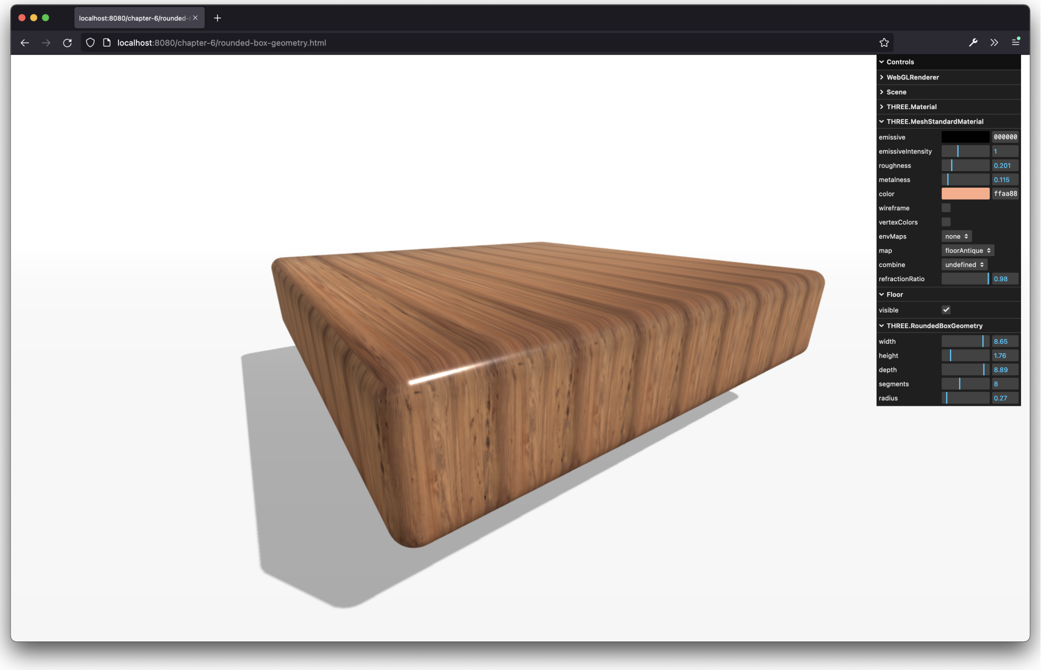
```
import { BoxLineGeometry } from 'three/examples/jsm/
  geometries/BoxLineGeometry'
const material = new THREE.LineBasicMaterial({ color:
  0x000000 }),
const geometry = new BoxLineGeometry(width, height, depth,
  widthSegments, heightSegments, depthSegments)
const lines = new THREE.LineSegments(geometry, material)
scene.add(lines)
```

For an explanation of the properties you can pass into this geometry, refer to the `THREE.BoxGeometry` section of *Chapter 5*.

Three.js also provides a slightly more advanced `THREE.BoxGeometry`, where you can have nicely rounded corners. You can do this with `RoundedBoxGeometry`.

THREE.RoundedBoxGeometry

This geometry uses the same properties as `THREE.BoxGeometry`, but it also allows you to specify how round the corners should be. In the `rounded-box-geometry` example, you can see how this looks:


localhost:8080/chapter-6/rounded-
localhost:8080/chapter-6/rounded-box-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.RoundedBoxGeometry

emissive 000000
emissiveIntensity 1
roughness 0.201
metalness 0.115
color ffaa88
wireframe ☐
vertexColors ☐
envMaps none
map floorAntique
combine undefined
refractionRatio 0.98
visible ☑
width 8.65
height 1.76
depth 8.89
segments 8
radius 0.27

Figure 6.4 – A box with rounded corners

For this geometry, we can specify the dimensions of the box by specifying `width`, `height`, and `depth`. Besides these properties, this geometry provides two additional ones:

- `radius`: This is the size of the rounded corners. The higher this value, the more rounded the corners will be.
- `segments`: This property defines how detailed the corners will be. If this is set to a low value, Three.js will use fewer vertices for the definition of the rounded corners.

Before we move on to showing how you can create 3D geometries from a 2D object, we'll look at the final geometry provided by Three.js, `TeapotGeometry`.

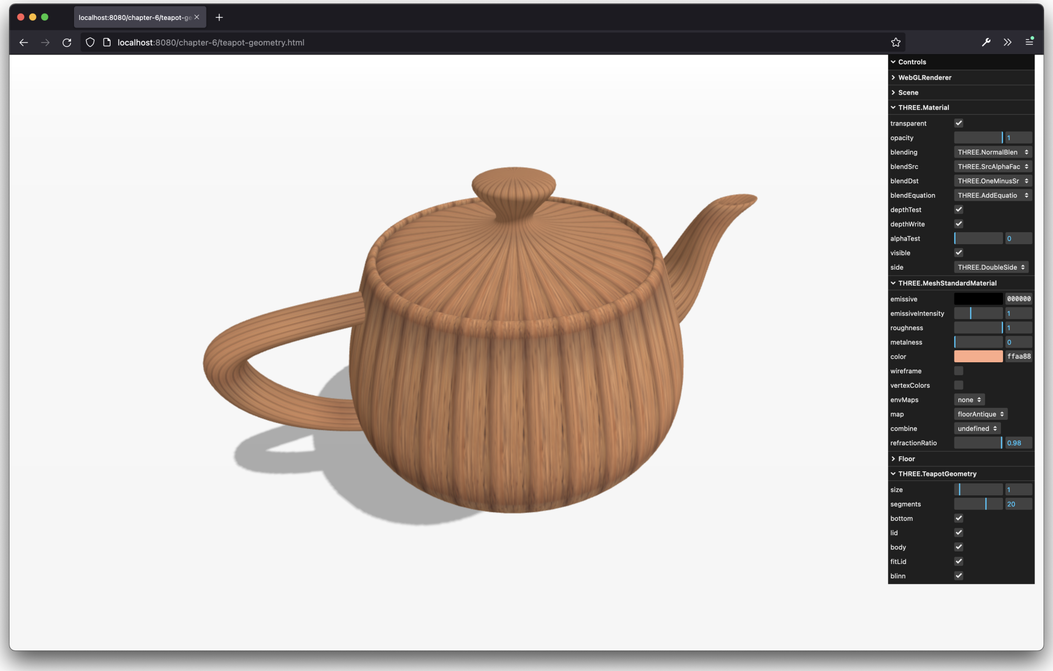
TeapotGeometry

`TeapotGeometry` is a geometry that you can use to render, not very surprisingly, a teapot. This teapot is a standard reference model for 3D renders and has been used since 1975. More information on the history of this model can be found here: `https://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206`.

Using this model works in exactly the same way as all the other models we've seen so far:

```
import { TeapotGeometry } from 'three/examples/jsm/
  geometries/TeapotGeometry'
...
const geom = new TeapotGeometry(size, segments, bottom,
  lid, body, fitLid, blinn)
```

You specify the specific properties and then create the geometry, which you assign to `THREE.Mesh`. The result, depending on the properties, looks like this (in the `teapot-geometry.html` example):


localhost:8080/chapter-6/teapot-ge
localhost:8080/chapter-6/teapot-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material

transparent ☑
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.DoubleSide

THREE.MeshStandardMaterial

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color ffaa88
wireframe ☐
vertexColors ☐
envMaps none
map floorAntique
combine undefined
refractionRatio 0.98

Floor
THREE.TeapotGeometry

size 1
segments 20
bottom ☑
lid ☑
body ☑
fitLid ☑
blinn ☑

Figure 6.5 – The Utah teapot

To configure this geometry, you can use the following properties:

- `size`: This is the size of the teapot.
- `segments`: This defines how many segments are used to create the wireframe of this teapot. The more segments you use, the more smooth the teapot will look.
- `bottom`: If set to `true`, the bottom of the teapot will be rendered. If `false`, the bottom won't be rendered, which you could use when the teapot is located on a surface and there is no need to render the bottom of it.
- `lid`: If set to `true`, the lid of the teapot will be rendered. If `false`, the lid won't be rendered.
- `body`: If set to `true`, the body of the teapot will be rendered. If `false`, the body won't be rendered.
- `fitLid`: If set to `true`, the lid will exactly fit the teapot. If `false`, there will be a small space between the lid and the body of the teapot.
- `blinn`: This defines whether to use the same aspect ratio of the teapot as the original 1975 model this teapot is based on.

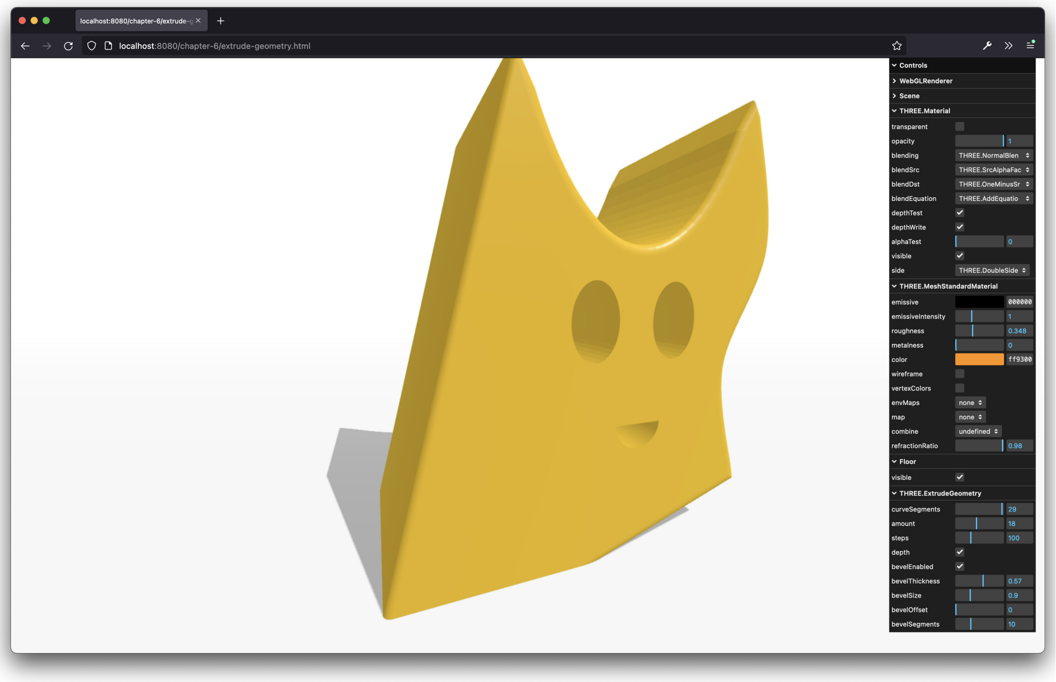
In the next sections, we'll look at an alternative way of creating geometries by extracting a 3D geometry from a 2D shape.

Creating a geometry by extruding a 2D shape

Three.js provides a way in which we can extrude a 2D shape into a 3D shape. By extruding, we mean stretching out a 2D shape along its *z* axis to convert it to 3D. For instance, if we extrude `THREE.CircleGeometry`, we get a shape that looks like a cylinder, and if we extrude `THREE.PlaneGeometry`, we get a cube-like shape. The most versatile way of extruding a shape is using `THREE.ExtrudeGeometry`.

THREE.ExtrudeGeometry

With `THREE.ExtrudeGeometry`, you can create a 3D object from a 2D shape. Before we dive into the details of this geometry, let's first look at an example, `extrude-geometry.html`. The following screenshot taken from the example shows this geometry:


localhost:8080/chapter-6/extrude-
localhost:8080/chapter-6/extrude-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.ExtrudeGeometry

transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.DoubleSide
emissive 000000
emissiveIntensity 1
roughness 0.348
metalness 0
color ff9300
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☑
curveSegments 29
amount 18
steps 100
depth ☑
bevelEnabled ☑
bevelThickness 0.57
bevelSize 0.9
bevelOffset 0
bevelSegments 10

Figure 6.6 – Creating a 3D geometry from a 2D shape

In this example, we took the 2D shape we created in the *2D geometries* section in *Chapter 5*, and used `THREE.ExtrudeGeometry` to convert it to 3D. As you can see in the preceding screenshot, the shape is extruded along the *z* axis, which results in a 3D shape. The code to create `THREE.ExtrudeGeometry` is very easy:

```
const geometry = new THREE.ExtrudeGeometry(drawShape(), {
    curveSegments,
    steps,
    depth,
    bevelEnabled,
    bevelThickness,
    bevelSize,
    bevelOffset,
    bevelSegments,
    amount
  })
```

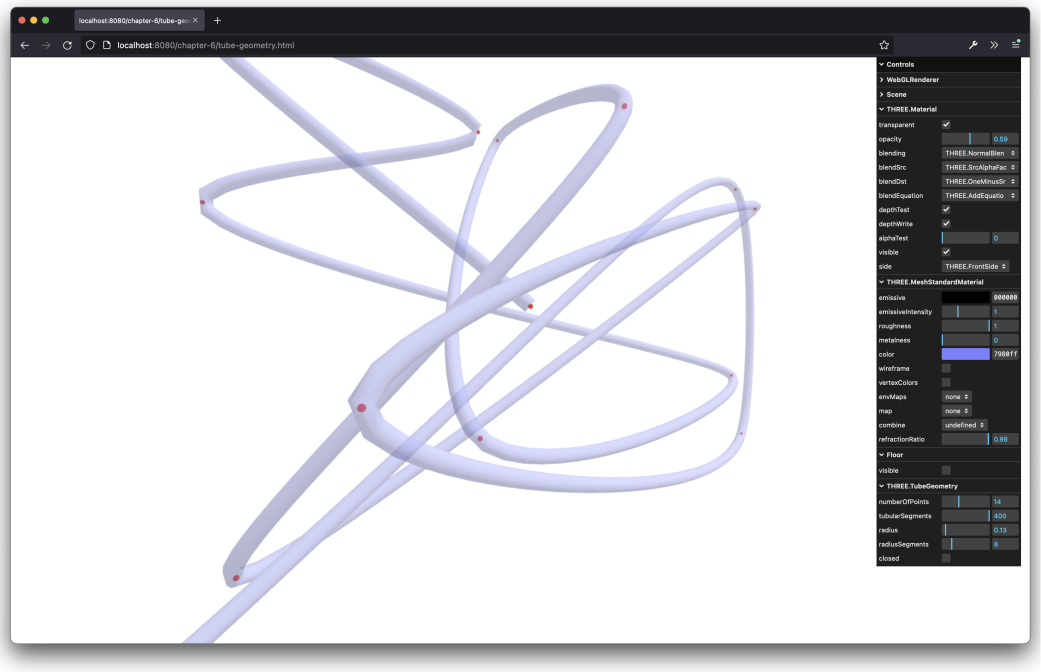
In this code, we created the shape with the `drawShape()` function, just as we did in *Chapter 5*. This shape is passed on to the `THREE.ExtrudeGeometry` constructor together with a set of properties. With these properties, you can define exactly how the shape should be extruded. The following list explains the options you can pass into `THREE.ExtrudeGeometry`:

- `shapes`: One or more shapes (`THREE.Shape` objects) are required to extrude the geometry. See *Chapter 5*, on how to create such a shape.
- `depth`: This determines how far the shape should be extruded (the depth). The default value is 100.
- `bevelThickness`: This determines the depth of the bevel. The bevel is the rounded corner between the front and back faces and the extrusion. This value defines how deep into the shape the bevel goes. The default value is `6`.
- `bevelSize`: This determines the height of the bevel. This is added to the normal height of the shape. The default value is `bevelThickness - 2`.
- `bevelSegments`: This defines the number of segments that will be used by the bevel. The more the number of segments used, the smoother the bevel will look. The default value is `3`. Note that if you add more segments, you're also increasing the vertex count, which could have an adverse effect on performance.
- `bevelEnabled`: If this is set to `true`, a bevel is added. The default value is `true`.
- `bevelOffset`: The distance from the outline of the shape where the bevel starts. The default value is `0`.
- `curveSegments`: This determines how many segments will be used when extruding the curves of shapes. The higher the number of segments used, the smoother the curves will look. The default value is `12`.
- `steps`: This defines the number of segments the shape will be divided into along the extrusion depth. The default value is `1`, which means it will have a single segment along its depth, without unnecessary additional vertices.
- `extrudePath`: This is the path (`THREE.CurvePath`) along which the shape should be extruded. If this isn't specified, the shape is extruded along the *z* axis. Note that if you've got a curving path, you also need to make sure to set a higher value for the `steps` property so that it can follow the curve accurately.
- `uvGenerator`: When you use a texture with your material, the UV mapping determines what part of a texture is used for a specific face. With the `uvGenerator` property, you can pass in your own object, which will create the UV settings for the faces that are created for the passed-in shapes. More information on UV settings can be found in *Chapter 10*, *Loading and Working with Textures*. If nothing is specified, `THREE.ExtrudeGeometry.WorldUVGenerator` is used.

If you want to use a different material for the faces and the sides, you can pass in an array of materials to `THREE.Mesh`. The first material passed in will be applied to the face, and the second material will be used for the sides. You can experiment with these options using the menu from the `extrude-geometry.html` example. In this example, we extruded the shape along its *z* axis. As you can see in the options listed earlier in this section, you can also extrude a shape along a path with the `extrudePath` option. In the following geometry, `THREE.TubeGeometry`, we'll do just that.

THREE.TubeGeometry

`THREE.TubeGeometry` creates a tube that extrudes along a 3D spline. You specify the path using a number of vertices, and `THREE.TubeGeometry` will create the tube. An example that you can experiment with can be found in the sources for this chapter (`tube-geometry.html`). The following screenshot shows this example:


localhost:8080/chapter-6/tube-geo
localhost:8080/chapter-6/tube-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.TubeGeometry

transparent ☑
opacity 0.59
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 7980ff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98
visible ☐
numberOfPoints 14
tubularSegments 400
radius 0.13
radiusSegments 8
closed ☐

Figure 6.7 – TubeGeometry based on random 3D vertices

As you can see in this example, we generate a number of random points and use those points to draw the tube. With the controls in the menu, we can define how the tube looks. The code needed to create a tube is very simple, as follows:

```
const points = ... // array of THREE.Vector3 objects
const tubeGeometry = new TubeGeometry(
  new THREE.CatmullRomCurve3(points),
  tubularSegments,
  radius,
  radiusSegments,
  closed
)
```

What we need to do first is get a set of vertices (the `points` variable) of the `THREE.Vector3` type, just like we did for `THREE.ConvexGeometry` and `THREE.LatheGeometry`. Before we can use these points, however, to create the tube, we first need to convert these points to `THREE.Curve`. In other words, we need to define a smooth curve through the points we defined. We can do this simply by passing in the array of vertices to the constructor of `THREE.CatmullRomCurve3`, or any of the other `Curve` implementations provided by Three.js. With this curve and the other arguments (which we'll explain in this section), we can create the tube and add it to the scene.

In this example, we've used `THREE.CatmullRomCurve3`. Three.js provides a number of other curves you can use as well that take slightly different arguments, but they can be used to create different curve implementations. Out of the box, Three.js comes with the following curves: `ArcCurve`, `CatmullRomCurve3`,`CubicBezierCurve`,`CubicBezierCurve3`,`EllipseCurve`,`LineCurve`,`LineCurve3`,`QuadraticBezierCurve`, `QuadraticBezierCurve3`, and `SplineCurve`.

`THREE.TubeGeometry` takes some other arguments besides the curve. The following lists all the arguments for `THREE.TubeGeometry`:

- `path`: This is `THREE.SplineCurve3`, which describes the path this tube should follow.
- `tubularSegments`: These are the segments used to build up the tube. The default value is `64`. The longer the path, the more segments you should specify.
- `radius`: This is the radius of the tube. The default value is `1`.
- `radiusSegments`: This is the number of segments to be used along the length of the tube. The default value is `8`. The more you use, the more round the tube will look.
- `closed`: If this is set to `true`, the start and the end of the tube will be connected. The default value is `false`.

The last extrude example we'll show in this chapter isn't really a different type of geometry, but we'll use `THREE.ExtrudeGeometry` to create extrusions from an SVG image.

What is SVG?

SVG is an XML-based standard that can be used to create vector-based 2D images for the web. This is an open standard that is supported by all modern browsers. Directly working with SVG and manipulating it from JavaScript, however, isn't very straightforward. Luckily, there are a couple of open source JavaScript libraries that make working with SVG a lot easier. `Paper.js`, `Snap.js`, `D3.js`, and `Raphael.js` are some of the best. If you want a graphical editor, you can also use the open source Inkscape product.

Extruding 3D shapes from an SVG element

When we discussed `THREE.ShapeGeometry` in *Chapter 5*, we mentioned that SVG follows pretty much the same approach to drawing shapes. In this section, we'll look at how you can use SVG images together with `THREE.SVGLoader` to extrude SVG images. We'll use the Batman logo as an example:

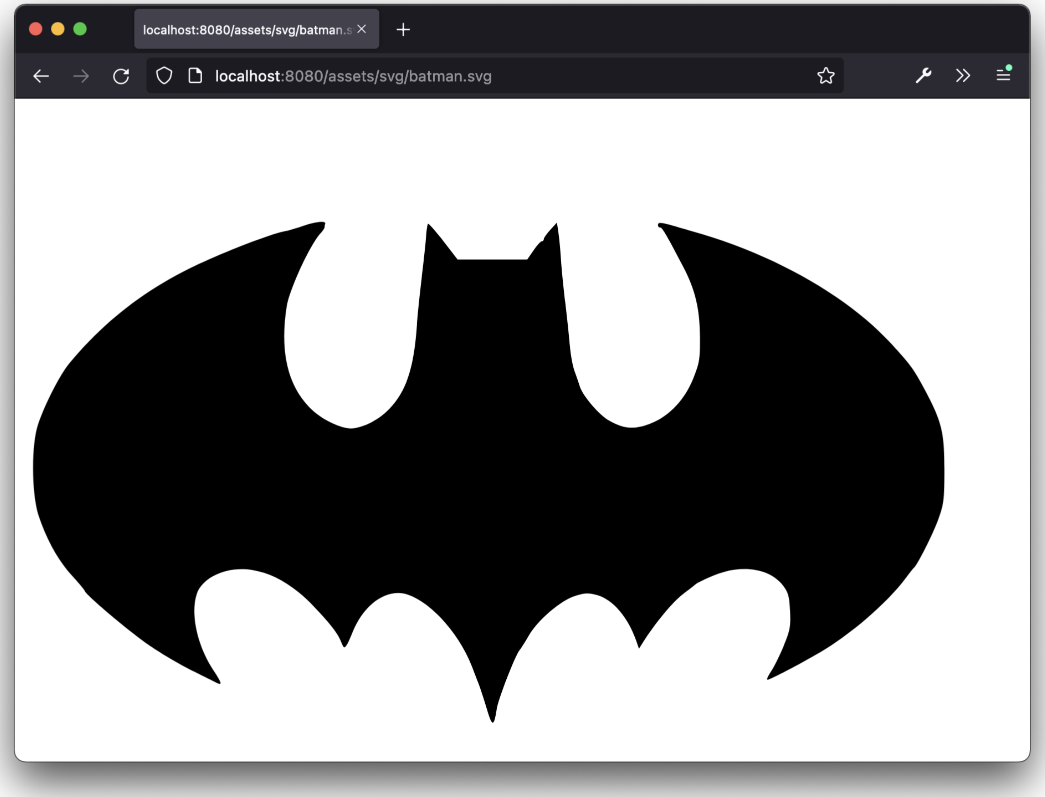

localhost:8080/assets/svg/batman.s
localhost:8080/assets/svg/batman.svg

Figure 6.8 – The Batman SVG base image

First, let's look at what the original SVG code looks like (you can also see this for yourself when looking at the source code of the `assets/svg/batman.svg` file):

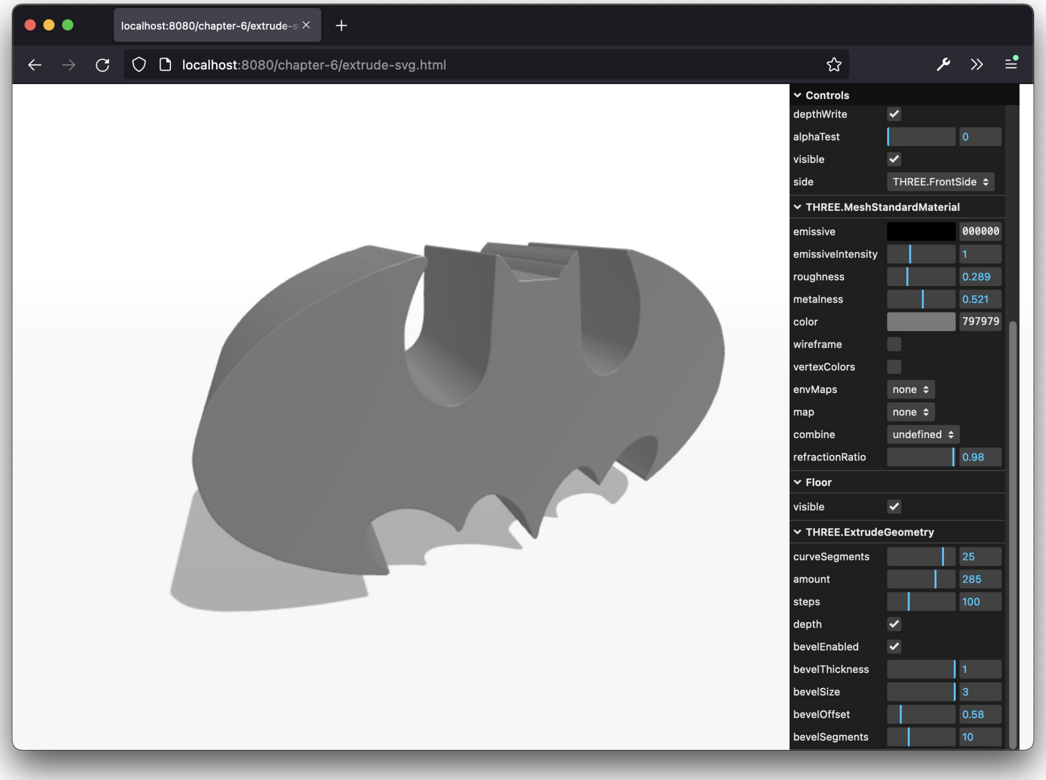
```
<svg version="1.0" xmlns="http://www.w3.org/2000/
svg"   xmlns:xlink="http://www.w3.org/1999/xlink" x="0px"
y="0px" width="1152px" height="1152px" xml:space="preserve">
  <g>
    <path   id="batman-path" style="fill:rgb(0,0,0);" d="M
261.135 114.535 C 254.906 116.662 247.491 118.825 244.659
119.344 C
    229.433 122.131 177.907 142.565 151.973 156.101 C   111.417
    177.269 78.9808 203.399 49.2992 238.815 C 41.0479   248.66
    26.5057 277.248 21.0148 294.418 C 14.873 313.624    15.3588
    357.341 21.9304 376.806 C 29.244 398.469 39.6107    416.935
    52.0865 430.524 C 58.2431 437.23 63.3085 443.321    63.3431
    444.06 ... 261.135 114.535 "/>
  </g>
</svg>
```

Unless you're an SVG guru, this probably won't mean too much to you. Basically though, what you see here is a set of drawing instructions. For instance, `C 277.987 119.348 279.673 116.786 279.673 115.867` tells the browser to draw a cubic Bezier curve, and `L 489.242 111.787` tells us that we should draw a line to that specific position. Luckily though, we won't have to write the code to interpret this ourselves and can use `THREE.SVGLoader` instead, as you can see in the following code:

```
// returns a promise
const batmanShapesPromise = new SVGLoader().loadAsync('/assets/
svg/batman.svg')
// when promise resolves the svg will contain the shapes
batmanShapes.then((svg) => {
  const shapes = SVGLoader.createShapes(svg.paths[0])
  // based on the shapes we can create an extrude geometry
    as we've seen earlier
  const geometry = new THREE.ExtrudeGeometry(shapes, {
    curveSegments,
    steps,
    depth,
    bevelEnabled,
    bevelThickness,
    bevelSize,
    bevelOffset,
    bevelSegments,
    amount
  })
  ...
}
```

In this code fragment, you can see that we use `SVGLoader` to load the SVG file. We use `loadAsync` here, which will return a JavaScript `Promise`. When that `Promise` resolves, we get access to the loaded `svg` data. This data can contain a list of `path` elements, each representing the `path` element of the original SVG. In our example, we've only got one, so we use `svg.paths[0]` and pass it into `SVGLoader.createShapes` to convert it into an array of `THREE.Shape` objects. Now that we've got the shapes, we can use the same approach we used earlier when we extruded our custom-created 2D geometry and use `THREE.ExtrudeGeometry` to create a 3D model from the 2D-loaded SVG shapes.

The final result can be seen when you open the `extrude-svg.html` example in the browser:


localhost:8080/chapter-6/extrude-svg.html

Controls
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide

THREE.MeshStandardMaterial
emissive 000000
emissiveIntensity 1
roughness 0.289
metalness 0.521
color 797979
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
refractionRatio 0.98

Floor
visible ☑

THREE.ExtrudeGeometry
curveSegments 25
amount 285
steps 100
depth ☑
bevelEnabled ☑
bevelThickness 1
bevelSize 3
bevelOffset 0.58
bevelSegments 10

Figure 6.9 – A 3D-created Batman logo extruded from a 2D SVG image

The last geometry we'll discuss in this section is `THREE.ParametricGeometry`. With this geometry, you can specify a couple of functions that are used to programmatically create geometries.

THREE.ParametricGeometry

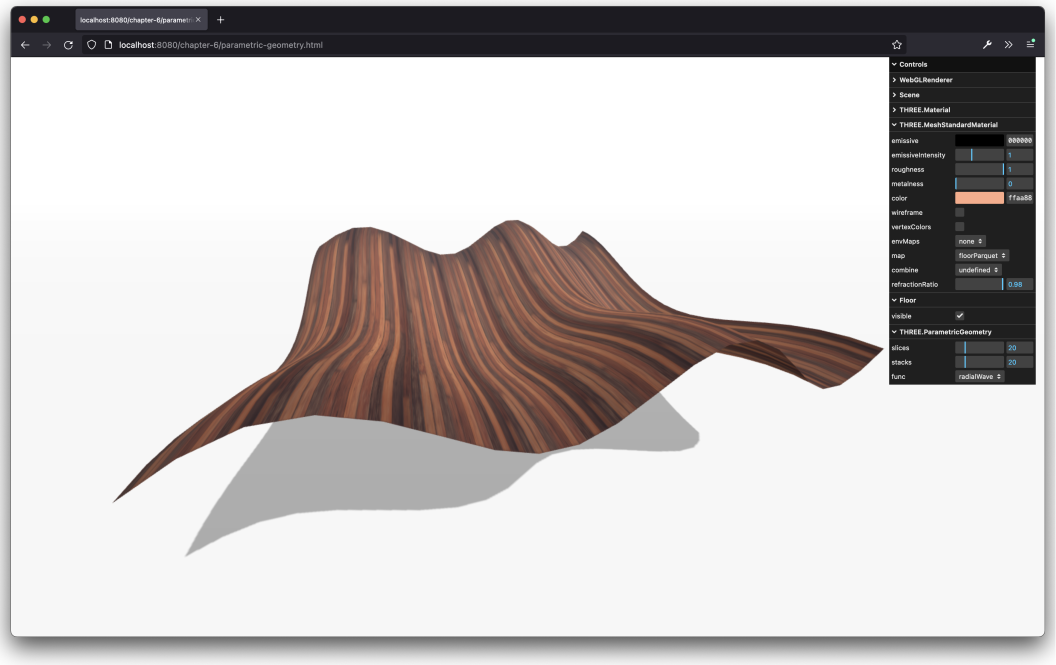
With `THREE.ParametricGeometry`, you can create a geometry based on an equation. Before we dive into our own example, a good thing to start with is to look at the examples already provided by Three.js. When you download the Three.js distribution, you get the `examples/js/ParametricGeometries.js` file. In this file, you can find a couple of examples of equations you can use together with `THREE.ParametricGeometry`.

The most basic example is the function to create a plane:

```
        plane: function ( width, height ) {
            return function ( u, v, target ) {
                const x = u * width;
                const y = 0;
                const z = v * height;
                target.set( x, y, z );
            };
        },
```

This function is called by `THREE.ParametricGeometry`. The `u` and `v` values will range from `0` to `1` and will be called a large number of times, for all the values from `0` to `1`. In this example, the `u` value is used to determine the `x` coordinate of the vector, and the `v` value is used to determine the `z` coordinate. When this is run, you'll have a basic plane with a width of `width` and a depth of `depth`.

In our example, we do something similar. However, instead of creating a flat plane, we create a wave-like pattern, as you can see in the `parametric-geometry.html` example. The following screenshot shows this example:


localhost:8080/chapter-6/parametri
localhost:8080/chapter-6/parametric-geometry.html

Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color ffaa88
wireframe ☐
vertexColors ☐
envMaps none
map floorParquet
combine undefined
refractionRatio 0.98
Floor
visible ☑
THREE.ParametricGeometry
slices 20
stacks 20
func radialWave

Figure 6.10 – A wave-like plane using a parametric geometry

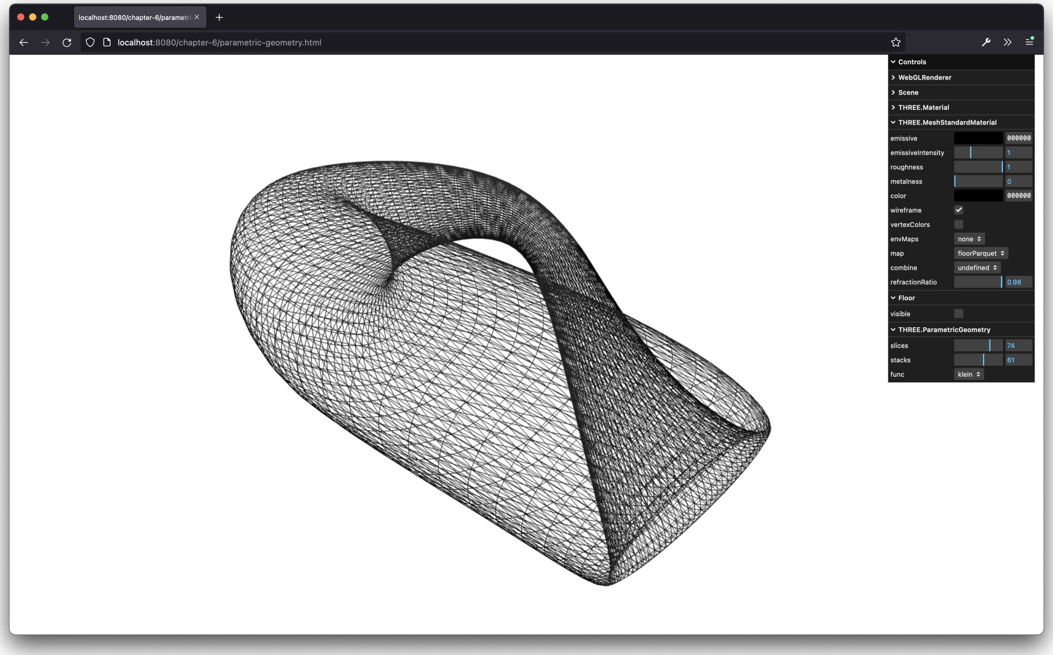
To create this shape, we passed the following function to `THREE.ParametricGeometry`:

```
const radialWave = (u, v, optionalTarget) => {
  var result = optionalTarget || new THREE.Vector3()
  var r = 20
  var x = Math.sin(u) * r
  var z = Math.sin(v / 2) * 2 * r + -10
  var y = Math.sin(u * 4 * Math.PI) + Math.cos(v * 2 *
    Math.PI)
  return result.set(x, y, z)
}
const geom = new THREE.ParametricGeometry(radialWave, 120,
  120);
```

As you can see in this example, with a few lines of code, we can create some really interesting geometries. In this example, you can also see the arguments we can pass to `THREE.ParametricGeometry`:

- `function`: This is the function that defines the position of each vertex based on the `u` and `v` values provided
- `slices`: This defines the number of parts the `u` value should be divided into
- `stacks`: This defines the number of parts the `v` value should be divided into

By changing the function, we can easily use the exact same approach to render a completely different object:


localhost:8080/chapter-6/parametri
localhost:8080/chapter-6/parametric-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.ParametricGeometry

emissive 000000
emissiveIntensity 1
roughness 1
metalness 0
color 000000
wireframe ☑
vertexColors ☐
envMaps none
map floorParquet
combine undefined
refractionRatio 0.98
visible ☐
slices 74
stacks 61
func klein

Figure 6.11 – A Klein bottle rendered using a parametric geometry

Here's a final note on how to use the `slices` and `stacks` properties before moving on to the next part of this chapter. We mentioned that the `u` and `v` properties are passed into the function argument provided and that the values of these two properties range from `0` to `1`. With the `slices` and `stacks` properties, we can define how often the passed-in function is called. If, for instance, we set `slices` to `5` and `stacks` to `4`, the function will be called with the following values:

```
u:0/5, v:0/4
u:1/5, v:0/4
u:2/5, v:0/4
u:3/5, v:0/4
u:4/5, v:0/4
u:5/5, v:0/4
u:0/5, v:1/4
u:1/5, v:1/4
...
u:5/5, v:3/4
u:5/5, v:4/4
```

So, the higher these values are, the more vertices you get to specify and the smoother your created geometry will be. You can use the menu at the right of the `parametric-geometry.html` example to see this effect.

For more examples, you can look at the `examples/js/ParametricGeometries.js` file in the Three.js distribution. This file contains functions to create the following geometries:

- Klein bottle
- Plane
- Flat Mobius strip
- 3D Mobius strip
- Tube
- Torus knot
- Sphere
- Plane

Sometimes, you need to see more details about your geometry, and you don't care too much about materials and how the mesh will be rendered. If you want to look at the vertices and the faces, or even just the outline, Three.js provides a couple of geometries that can help you with this (besides enabling the `wireframe` property of the material you use for the mesh). We'll explore these in the following section.

Geometries you can use for debugging

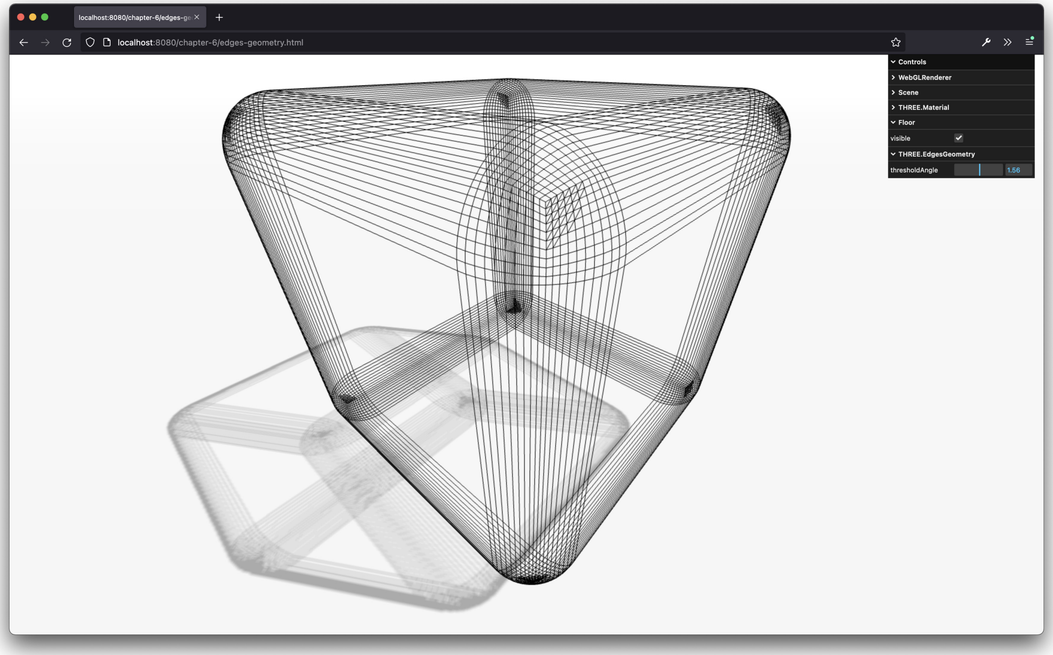
Three.js comes with two helper geometries out of the box that make it easier to see the details or just the outline of a geometry:

- `THREE.EdgesGeometry`, which provides a geometry that only renders the edges of a geometry
- `THREE.WireFrameGeometry`, which renders just the geometry without showing any faces

First, let's look at `THREE.EdgesGeometry`.

THREE.EdgesGeometry

With `THREE.EdgesGeometry`, you wrap an existing geometry, which is then rendered by just showing the edges and not the individual vertices and faces. An example of this is shown in the `edges-geometry.html` example:


localhost:8080/chapter-6/edges-ge
localhost:8080/chapter-6/edges-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
Floor
THREE.EdgesGeometry

visible ☑

thresholdAngle 1.56

Figure 6.12 – EdgesGeometry only showing the edges, not the individual faces

In the previous screenshot, you can see that the outline of `RoundedBoxGeometry` is shown, where we just see the edges. Since `RoundedBoxGeometry` has smooth corners, those are shown when using `THREE.EdgesGeometry`.

To use this geometry, you just wrap an existing geometry like this:

```
const baseGeometry = new RoundedBoxGeometry(3, 3, 3, 10, 0.4)
const edgesGeometry = THREE.EdgesGeometry(baseGeometry, 1.5)
}
```

The only property `THREE.EdgesGeometry` takes is `thresholdAngle`. With this property, you can determine when this geometry draws an edge. In `edges-geometry.html`, you can control this property to see the effect.

If you've got an existing geometry and want to see the wireframe, you can configure a material to show this wireframe:

```
const material = new THREE.MeshBasicMaterial({ color: 0xffff00,
wireframe: true })
```

Three.js also provides a different way of using `THREE.WireFrameGeometry`.

THREE.WireFrameGeometry

This geometry simulates the behavior you see when you set the `wireframe` property of a material to `true`:

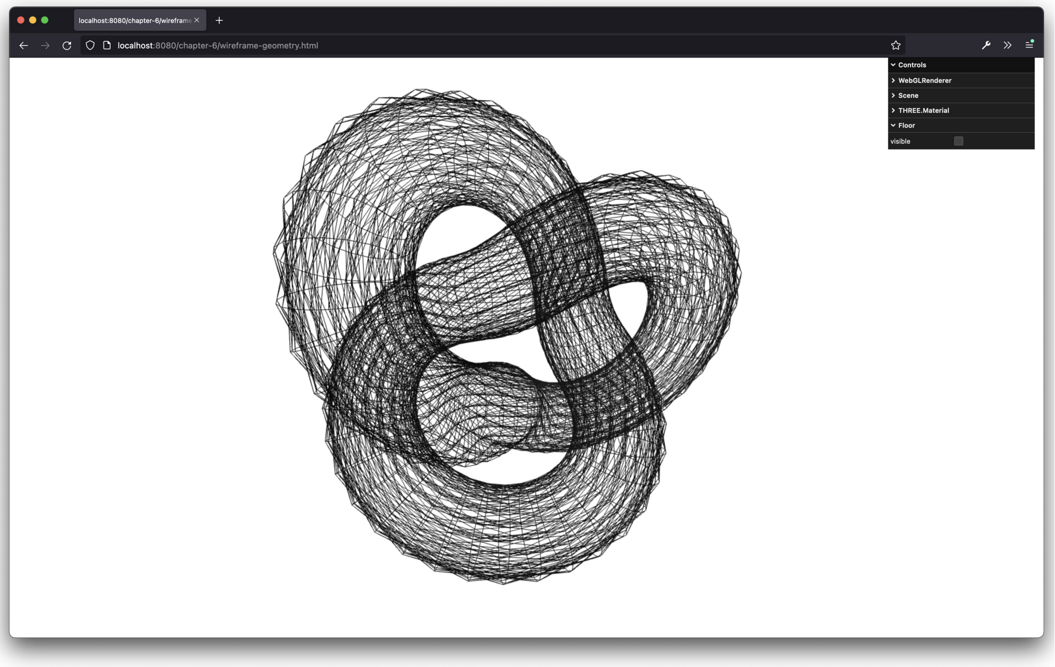

localhost:8080/chapter-6/wirefram
localhost:8080/chapter-6/wireframe-geometry.html
Controls
WebGLRenderer
Scene
THREE.Material
Floor
visible

Figure 6.13 – Wireframe geometry showing all the individual faces of a geometry

Using this material works in the same way as using `THREE.EdgesGeometry`:

```
const baseGeometry = new THREE.TorusKnotBufferGeometry(3, 1,
100, 20, 6, 9)
const wireframeGeometry = new THREE.
WireframeGeometry(baseGeometry)
```

This geometry doesn't take any additional properties.

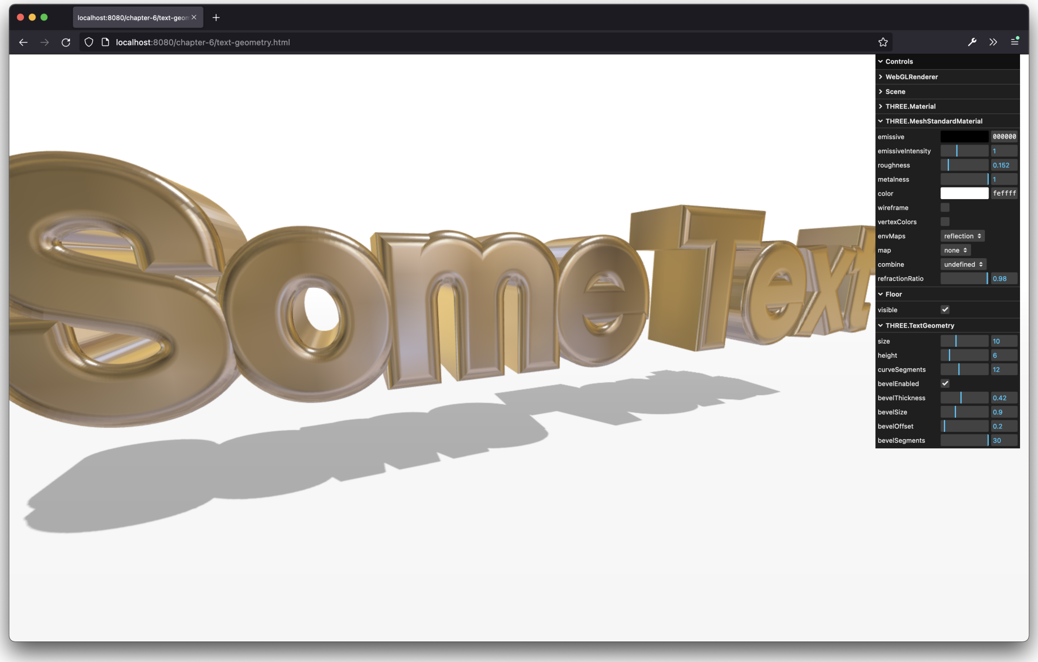
The last part of this chapter deals with creating 3D text objects. We'll show you two different approaches, one with the `THREE.Text` object and one where we use an external library.

Creating a 3D text mesh

In this section, we'll have a quick look at how you can create 3D text. First, we'll look at how to render text using the fonts provided by Three.js, and how you can use your own fonts for this. Then, we'll show a quick example of using an external library called Troika (`https://github.com/protectwise/troika`) that makes it really easy to create labels and 2D text elements and add them to your scene.

Rendering text

Rendering text in Three.js is very easy. All you have to do is define the font you want to use and use the same extrude properties we saw when we discussed `THREE.ExtrudeGeometry`. The following screenshot shows a `text-geometry.html` example of how to render text in Three.js:


localhost:8080/chapter-6/text-geom
localhost:8080/chapter-6/text-geometry.html

Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
emissive 000000
emissiveIntensity 1
roughness 0.152
metalness 1
color feffff
wireframe ☐
vertexColors ☐
envMaps reflection
map none
combine undefined
refractionRatio 0.98
Floor
visible ☑
THREE.TextGeometry
size 10
height 6
curveSegments 12
bevelEnabled ☑
bevelThickness 0.42
bevelSize 0.9
bevelOffset 0.2
bevelSegments 30

Figure 6.14 – Rendering text in Three.js

The code required to create this 3D text is as follows:

```
import { FontLoader } from 'three/examples/jsm/
  loaders/FontLoader'
import { TextGeometry } from 'three/examples/jsm/
  geometries/TextGeometry'
...
new FontLoader()
  .loadAsync('/assets/fonts/helvetiker_regular.typeface.json')
  .then((font) => {
      const textGeom =  new TextGeometry('Some Text', {
          font,
          size,
        height,
        curveSegments,
        bevelEnabled,
        bevelThickness,
        bevelSize,
        bevelOffset,
        bevelSegments,
        amount
   })
   ...
 )
```

In this code fragment, you can see that we first have to load the font. For this, Three.js provides `FontLoader()`, where we provide the name of the font to load, just like we did with `SVGLoader`, where we get back a JavaScript `Promise`. Once that `Promise` resolves, we use the loaded font to create `TextGeometry`.

The options we can pass into `THREE.TextGeometry` match those that we can pass into `THREE.ExtrudeGeometry`:

- `font`: The loaded font to use for the text.
- `size`: This is the size of the text. The default value is `100`.
- `height`: This is the length (depth) of the extrusion. The default value is `50`.
- `curveSegments`: This defines the number of segments used when extruding the curves of shapes. The more segments there are, the smoother the curves will look. The default value is `4`.
- `bevelEnabled`: If this is set to `true`, a bevel is added. The default value is `false`.
- `bevelThickness`: This is the depth of the bevel. The bevel is the rounded corner between the front and back faces and the extrusion. The default value is `10`.
- `bevelSize`: This is the height of the bevel. The default value is `8`.
- `bevelSegments`: This defines the number of segments that will be used by the bevel. The more segments there are, the smoother the bevel will look. The default value is `3`.
- `bevelOffset`: This is the distance from the outline of the shape where the bevel starts. The default value is `0`.

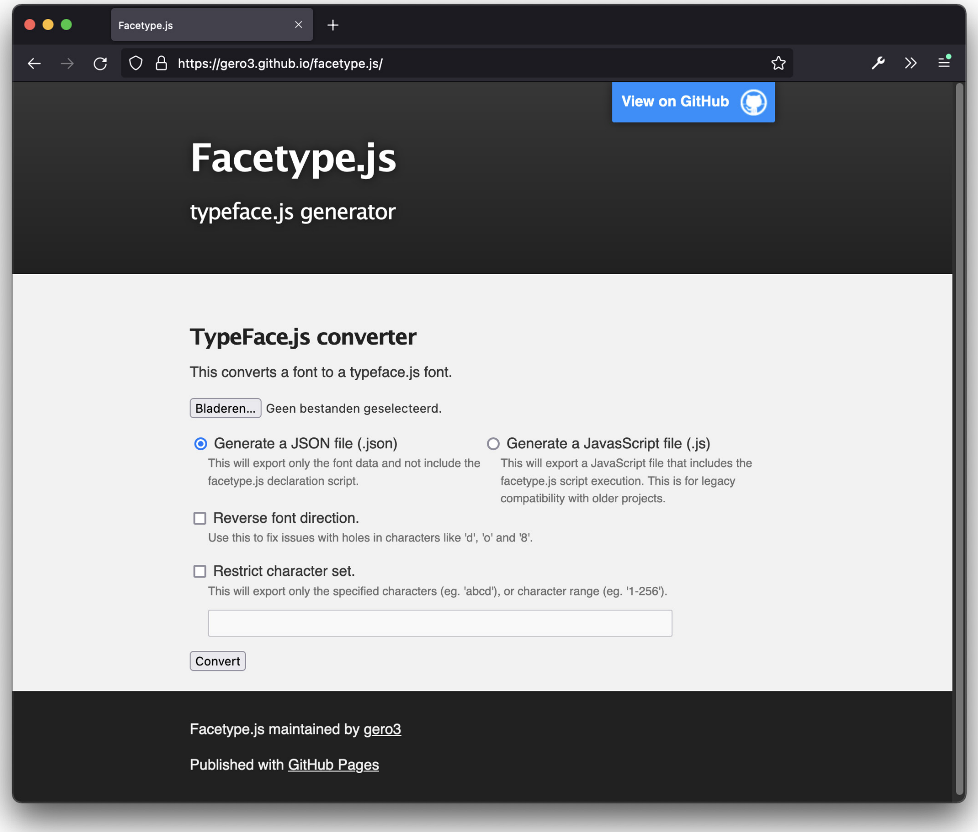
Since `THREE.TextGeometry` is also `THREE.ExtrudeGeometry`, the same approach applies if you want to use a different material for the front and the sides of the material. If you pass in an array of two materials when creating `THREE.Mesh`, Three.js will apply the first material to the front and the back of the text, and the second one to the sides.

It's also possible to use other fonts with this geometry, but you first need to convert them to JSON – how to do this is shown in the next section.

Adding custom fonts

There are a couple of fonts provided by Three.js that you can use in your scenes. These fonts are based on the fonts provided by the `TypeFace.js` library. `TypeFace.js` is a library that can convert TrueType and OpenType fonts to JavaScript. The resulting JavaScript file or JSON file can be included in your page, and the font can then be used in Three.js. In older versions, the JavaScript file was used, but in later Three.js versions, Three.js switched to using the JSON file.

To convert an existing OpenType or TrueType font, you can use the web page at `https://gero3.github.io/facetype.js/`:



Facetype.js

typeface.js generator

TypeFace.js converter

This converts a font to a typeface.js font.

Bladeren... Geen bestanden geselecteerd.

● Generate a JSON file (.json)
This will export only the font data and not include the facetype.js declaration script.

○ Generate a JavaScript file (.js)
This will export a JavaScript file that includes the facetype.js script execution. This is for legacy compatibility with older projects.

☐ Reverse font direction.
Use this to fix issues with holes in characters like 'd', 'o' and '8'.

☐ Restrict character set.
This will export only the specified characters (eg. 'abcd'), or character range (eg. '1-256').

Facetype.js maintained by gero3

Published with GitHub Pages

Figure 6.15 – Converting a font to a typeface-supported format

On this page, you can upload a font, and it will be converted to JSON for you. Note that this won't work so well for all types of fonts. The simpler the font (more straight lines), the better the chance that it will be rendered correctly when used in Three.js. The resulting file looks like this, where each of the characters (or glyphs) is described:

```
{"glyphs":{"¦":{"x_min":359,"x_max":474,"ha":836,"o":"m 474 971
l 474 457 l
359 457 l 359 971 l 474 971 m 474 277 l 474 -237 l 359 -237 l
359 277 l 474
277 "},"ž":{"x_min":106,"x_max":793,"ha":836,"o":"m 121 1013 l
778 1013 l
778 908 l 249 115 l 793 115 l 793 0 l 106 0 l 106 104 l 620 898
l 121 898 l
121 1013 m 353 1109 l 211 1289 l 305 1289 l 417 1168 l 530 1289
l 625 1289
l 482 1109 l 353 1109 "},"Á":{"x_min":25,"x_
max":811,"ha":836,"o":"m 417
892 l 27 ....
```

Once you've got the JSON file, you can use `FontLoader` (as we showed previously in the *Rendering text* section) to load this font and assign it to the `font` property of the options you can pass into `TextGeometry`.

For the final example of this chapter, we're going to look at a different way to create text with Three.js.

Creating text using the Troika library

If you want to create labels or 2D text marks for certain parts of your scene, there is an alternative option to using the `THREE.Text` geometry. You can also use an external library called Troika: `https://github.com/protectwise/troika`.

This is a fairly big library that provides lots of functionalities to add interactivity to your scenes. For this example, we'll only look at the text module of that library. An example of what we're going to create is shown in the `troika-text.html` example:

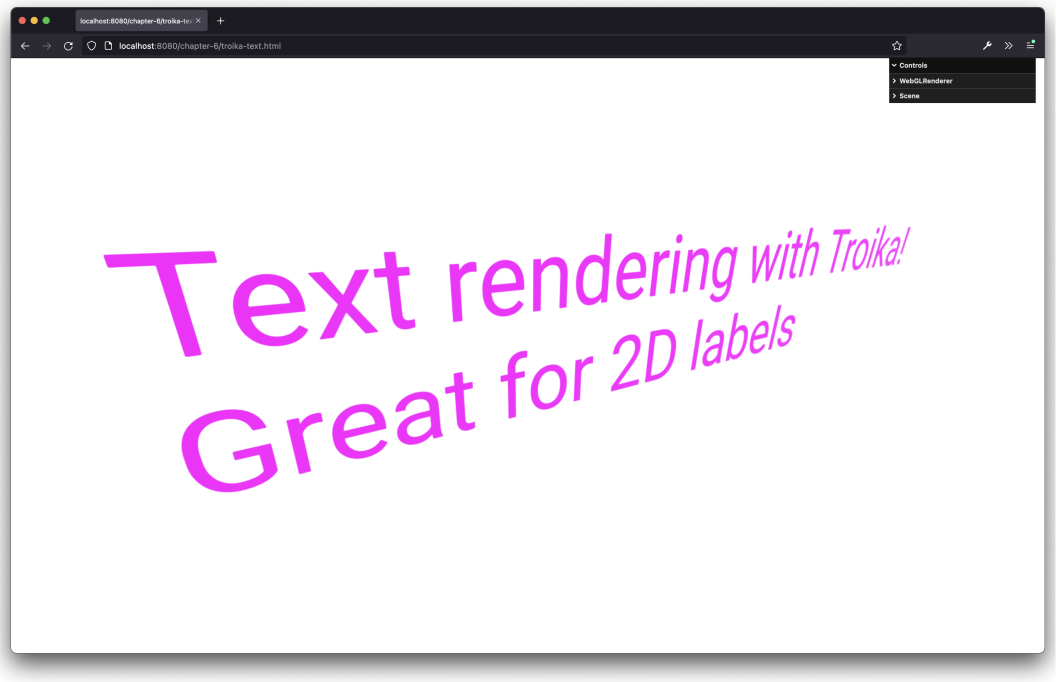

localhost:8080/chapter-6/troika-tex
localhost:8080/chapter-6/troika-text.html
Controls
WebGLRenderer
Scene
Text rendering with Troika!
Great for 2D labels

Figure 6.16 – Troika text for 2D labels

To use this library, we first have to install it (if you followed the instructions from *Chapter 1*, *Creating Your First 3D Scene with Three.js*, you can already use this library): `$ yarn add troika-three-text`. Once installed, we can import it and use it just like we do with the rest of the modules provided by Three.js:

```
import { Text } from 'troika-three-text'
const troikaText = new Text()
troikaText.text = 'Text rendering with Troika!\nGreat for
  2D labels'
troikaText.fontSize = 2
troikaText.position.x = -3
troikaText.color = 0xff00ff
troikaText.sync()
scene.add(troikaText)
```

In the previous code fragment, we showed how you can use Troika to create a simple text element. You only need to call the `Text()` constructor and set the properties. One thing to keep in mind, however, is that whenever you change a property in the `Text()` object, you have to call `troikaText.sync()`. That will make sure that the changes are also applied to the model rendered on the screen.

Summary

We saw a lot in this chapter. We introduced a couple of advanced geometries and showed you how you can create and render text elements with Three.js. We showed you how you can create really beautiful shapes using advanced geometries such as `THREE.ConvexGeometry`, `THREE.TubeGeometry`, and `THREE.LatheGeometry` and how can you experiment with these geometries to get the results you're looking for. A very nice feature is that we can also convert existing SVG paths to Three.js using `THREE.ExtrudeGeometry`.

We've also quickly looked at a couple of geometries that are very useful for debugging purposes. `THREE.EdgesGeometry` shows just the edges of another geometry, and `THREE.WireframeGeometry` can be used to show the wireframe of some other geometry.

Finally, if you want to create 3D text, Three.js provides `TextGeometry`, where you can pass in a font that you want to use. Three.js comes with a couple of fonts, but you can also create your own fonts. However, remember that complex fonts often won't convert correctly. An alternative to using `TextGeometry` is using the Troika library, which makes it very easy to create 2D text labels and place them anywhere in the scene.

Until now, we looked at solid (or wireframe) geometries, where vertices are connected to each other to form faces. In the upcoming chapter, we'll look at an alternative way of visualizing geometries using something called particles or points. With particles, we don't render complete geometries — we just render the individual vertices as points in space. This allows you to create great-looking 3D effects that perform well.

7

Points and Sprites

In the previous chapters, we discussed the most important concepts, objects, and APIs that Three.js has to offer. In this chapter, we'll look into the only concepts we've skipped until now: points and sprites. With `THREE.Points` (sometimes also called sprites), it is very easy to create many small rectangles that always face the camera and you can use to simulate rain, snow, smoke, and other interesting effects. For instance, you can render individual geometries as a set of points and control these points separately. In this chapter, we'll explore the various point- and sprite-related features provided by Three.js.

To be more specific, we'll look at the following topics in this chapter:

- Creating and styling particles using `THREE.SpriteMaterial` and `THREE.PointsMaterial`
- Using `THREE.Points` to create a group of points
- Using the canvas to style each point individually
- Using a texture to style the individual points
- Animating `THREE.Points` objects
- Creating a `THREE.Points` object from existing geometries

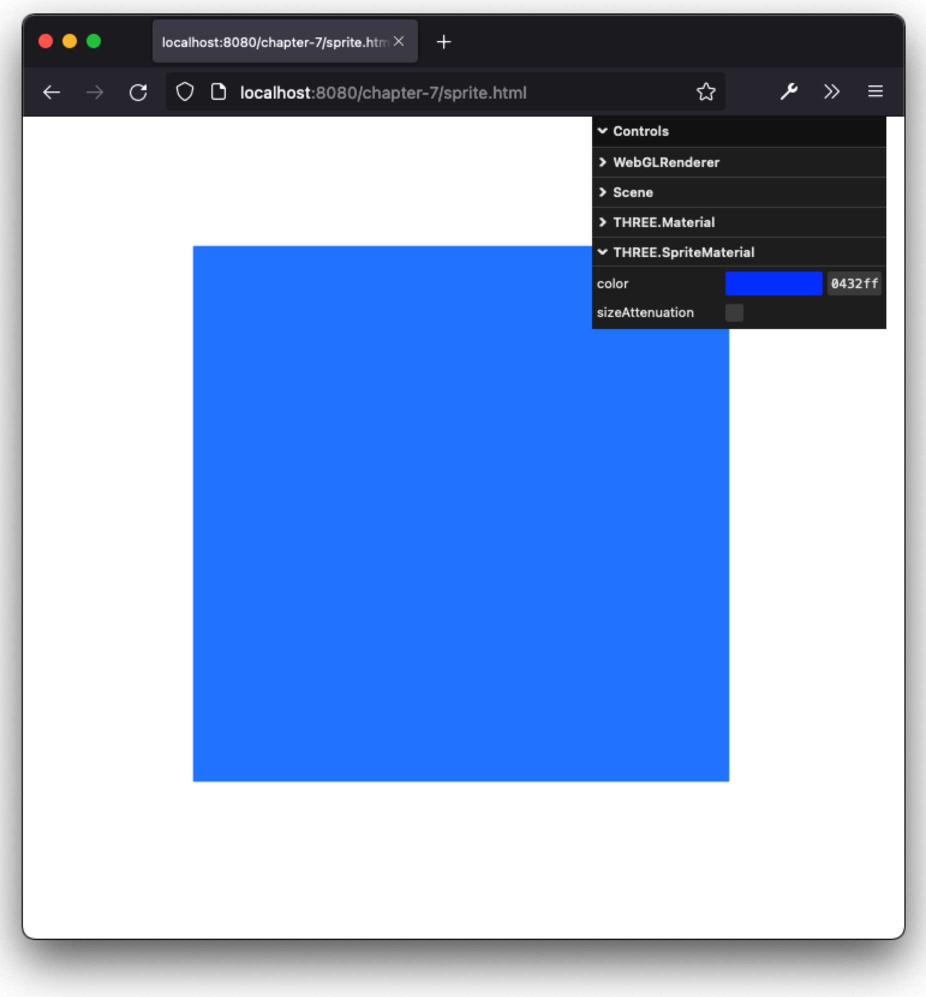
A quick note on some of the names used in this chapter

In newer versions of Three.js, the names of the objects related to points have changed several times. The `THREE.Points` object was previously named `THREE.PointCloud` and, in even older versions, it was called `THREE.ParticleSystem`. `THREE.Sprite` used to be called `THREE.Particle`, and the materials have also undergone several name changes. So, if you see online examples using these old names, remember that they are talking about the same concepts.

Let's start by exploring what a particle is and how you can create one.

Understanding points and sprites

As we do with most new concepts, we'll start with an example. In the sources for this chapter, you'll find an example called `sprite.html`. Upon opening this example, you'll see a minimalistic scene, containing a simple colored square:

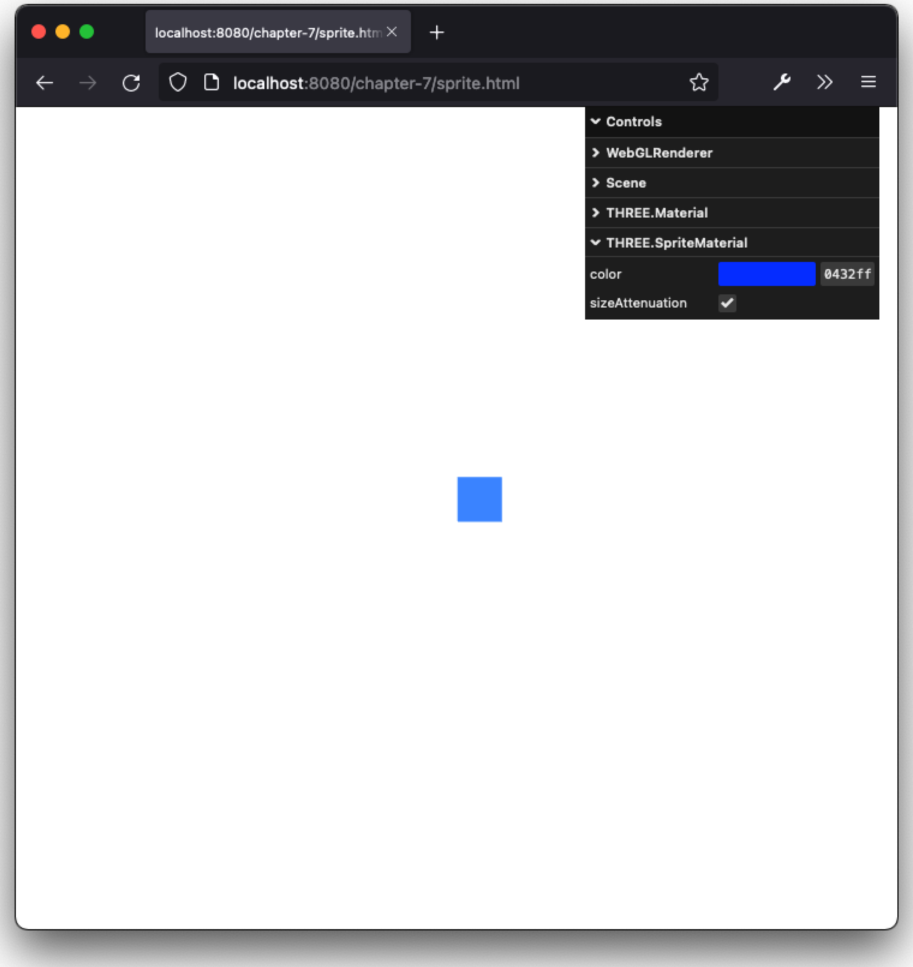

localhost:8080/chapter-7/sprite.htm
localhost:8080/chapter-7/sprite.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.SpriteMaterial

color 0432ff

sizeAttenuation ☐

Figure 7.1 – A single rendered sprite

You can use your mouse to rotate around this scene. One thing you'll notice is that no matter how you look at the square, it will always look the same. For instance, the following screenshot shows a view of the same scene from a different position:


localhost:8080/chapter-7/sprite.htm
localhost:8080/chapter-7/sprite.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.SpriteMaterial

color 0432ff
sizeAttenuation ✓

Figure 7.2 – A single rendered sprite will always be facing the camera

As you can see, the sprite is still angled toward the camera, and you can't look behind it. You can think of a sprite as a 2D plane that always faces the camera. If you create a sprite without any properties, they are rendered as small, white, two-dimensional squares. To create a sprite, we only need to provide a material:

```
const material = new THREE.SpriteMaterial({ size: 0.1,
  color: 0xff0000 })
const sprite = new THREE.Sprite(material)
sprite.position.copy(new THREE.Vector3(1,1,1))
```

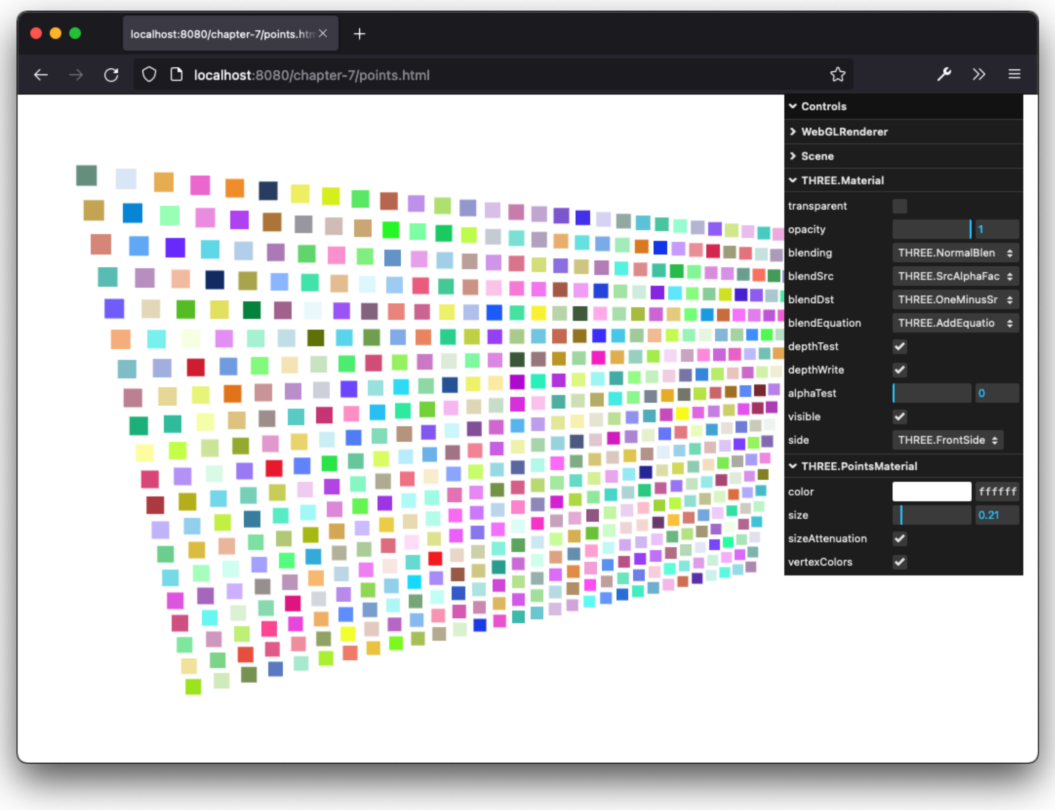
You can configure how the sprite appears using `THREE.SpriteMaterial`:

- `color`: This is the color of the sprite. The default color is white.
- `sizeAttenuation`: If this is set to `false`, the sprite will have the same size, regardless of how far from the camera it is positioned. If this is set to `true`, the size is based on the distance from the camera. The default value is `true`. Note that this only has an effect when using `THREE.PerspectiveCamera`. For `THREE.OrthographicCamera`, it always acts if set to `false`.
- `map`: With this property, you can apply a texture to the sprite. You can, for instance, make them look like snowflakes. This property isn't shown in this example but is explained in the *Styling particles using textures* section in this chapter.
- `opacity`: This, together with the `transparent` property, sets the opacity of the sprite. The default value is `1` (fully opaque).
- `transparent`: If this is set to `true`, the sprite will be rendered with the opacity set by the `opacity` property. The default value is `false`.
- `blending`: This is the blend mode to use when rendering the sprite.

Note that `THREE.SpriteMaterial` extends from the base `THREE.Material` object, so all the properties from that object also can be used on `THREE.SpriteMaterial`.

Before we move on to more interesting `THREE.Points` objects, let's look a bit closer at the `THREE.Sprite` object. A `THREE.Sprite` object extends from the `THREE.Object3D` object just as `THREE.Mesh` does. This means that most of the properties and functions you know from `THREE.Mesh` can be used on `THREE.Sprite`. You can set its position using the `position` attribute, scale it using the `scale` property, and move it along its axes using the `translate` property.

With `THREE.Sprite`, you can very easily create a set of objects and move them around the scene. This works well when you're working with a small number of objects, but you'll quickly run into performance issues when you want to work with a high number of `THREE.Sprite` objects. This is because each of the objects needs to be managed separately by Three.js. Three.js provides an alternative way of handling a large number of sprites using a `THREE.Points` object. With `THREE.Points`, Three.js doesn't have to manage many individual `THREE.Sprite` objects, just the `THREE.Points` instance. This will allow Three.js to optimize how it draws the sprites and will result in better performance. The following screenshot shows several sprites rendered with the `THREE.Points` object:


localhost:8080/chapter-7/points.htm
localhost:8080/chapter-7/points.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.PointsMaterial

transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color ffffff
size 0.21
sizeAttenuation ☑
vertexColors ☑

Figure 7.3 – Multiple points rendered from THREE.BufferGeometry

To create a `THREE.Points` object, we need to provide it with `THREE.BufferGeometry`. For the previous screenshot, we can create a `THREE.BufferGeometry` like this:

```
const createPoints = () => {
  const points = []
  for (let x = -15; x < 15; x++) {
    for (let y = -10; y < 10; y++) {
      let point = new THREE.Vector3(x / 4, y / 4, 0)
      points.push(point)
    }
  }
  const colors = new Float32Array(points.length * 3)
  points.forEach((e, i) => {
    const c = new THREE.Color(Math.random() * 0xffffff)
    colors[i * 3] = c.r
    colors[i * 3 + 1] = c.g
    colors[i * 3 + 2] = c.b
  })
  const geom = new THREE.BufferGeometry().setFromPoints(points)
  geom.setAttribute('color', new THREE.BufferAttribute(colors,
3, true))
  return geom
}
const material = new THREE.PointsMaterial({ size: 0.1,
  vertexColors: true, color: 0xffffff })
const points = new THREE.Points(createPoint(), material)
```

As you can see from this code fragment, first, we create an array of `THREE.Vector3` objects – one for each position where we want to create a sprite. Additionally, we set the `color` attribute on `THREE.BufferGeometry`, which is used to color each sprite. With `THREE.BufferGeometry` and an instance of `THREE.PointsMaterial`, we can create the `THREE.Points` object. The properties of `THREE.PointsMaterial` are pretty much the same as they are for `THREE.SpriteMaterial`:

- `color`: This is the color of the point. The default color is `0xffffff`.
- `sizeAttenuation` If this is set to `false`, all the points will have the same size, regardless of how far from the camera they are positioned. If this is set to `true`, the size is based on the distance from the camera. The default value is `true`.
- `map`: With this property, you can apply a texture to the point. You can, for instance, make them look like snowflakes. This property isn't shown in this example but is explained in the *Styling particles using textures* section later in this chapter.
- `opacity`: This, together with the `transparent` property, sets the opacity of the sprites. The default value is `1` (no opacity).
- `transparent`: If this is set to `true`, the sprites will be rendered with the opacity set by the `opacity` property. The default value is `false`.
- `blending`: This is the blend mode to use when rendering the sprites.
- `vertexColors`: Normally, all the points in `THREE.Points` have the same color. If this property is set to `true` and the color's buffer attribute has been set on the geometry, each point will take the color from that array. The default value is `false`.

As always, you can play around with these properties using the menu on the right in each of the examples.

So far, we've only rendered the particles as small squares, which is the default behavior. There are, however, two additional ways you can style particles, which we'll show in the next section.

Styling particles using textures

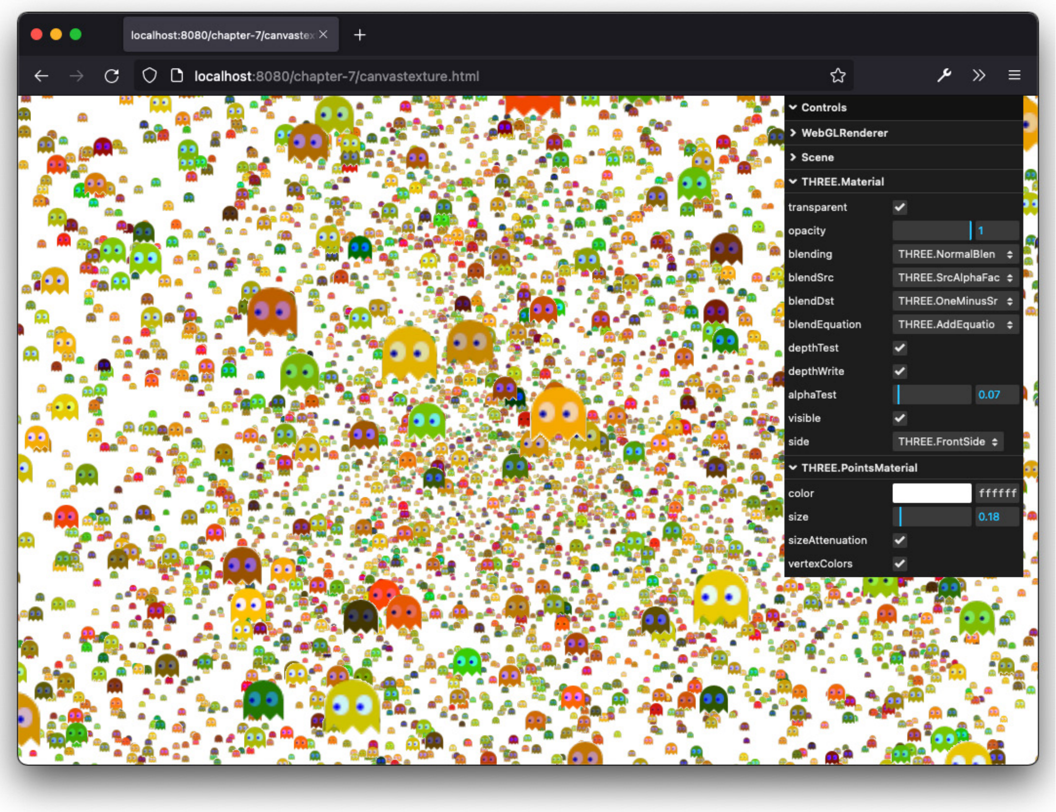
In this section, we'll look at the following two ways of changing what the sprite looks like:

- Use an HTML canvas to draw an image and show that for each sprite
- Load an external image file to define what each sprite looks like

Let's start by drawing the image ourselves.

Drawing an image on the canvas

In the attributes for `THREE.PointsMaterial`, we mentioned the `map` property. With the `map` property, we can load a texture for the individual points. With Three.js, this texture can also be the output from an HTML5 canvas. Before we look at the code, let's look at an example (`canvastexture.js`):


localhost:8080/chapter-7/canvastex
localhost:8080/chapter-7/canvastexture.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.PointsMaterial

transparent ✓
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ✓
depthWrite ✓
alphaTest 0.07
visible ✓
side THREE.FrontSide
color ffffff
size 0.18
sizeAttenuation ✓
vertexColors ✓

Figure 7.4 – Creating sprites using a canvas-based texture

Here, you can see that we've got a large set of Pac-Man-like ghosts on the screen. This uses the same approach that we saw in the *Understanding points and sprites* section earlier. This time, though, we aren't showing a simple square but an image. To create this texture, we can use the following code:

```
const createGhostTexture = () => {
  const canvas = document.createElement('canvas')
  canvas.width = 32
  canvas.height = 32
  const ctx = canvas.getContext('2d')
  // the body
  ctx.translate(-81, -84)
  ctx.fillStyle = 'orange'
  ctx.beginPath()
  ctx.moveTo(83, 116)
  ctx.lineTo(83, 102)
  ctx.bezierCurveTo(83, 94, 89, 88, 97, 88)
  // some code removed for clarity
  ctx.fill()
  // the eyes
  ctx.fillStyle = 'white'
  ctx.beginPath()
  ctx.moveTo(91, 96)
  ctx.bezierCurveTo(88, 96, 87, 99, 87, 101)
  ctx.bezierCurveTo(87, 103, 88, 106, 91, 106)
  // some code removed for clarity
  ctx.fill()
  // the pupils
  ctx.fillStyle = 'blue'
  ctx.beginPath()
  ctx.arc(101, 102, 2, 0, Math.PI * 2, true)
  ctx.fill()
  ctx.beginPath()
  ctx.arc(89, 102, 2, 0, Math.PI * 2, true)
  ctx.fill()
  const texture = new THREE.Texture(canvas)
  texture.needsUpdate = true
  return texture
}
```

As you can see, first, we create an HTML canvas, on which we start drawing using the various `ctx.` functions. In the end, we convert this canvas into a `THREE.Texture` by calling `new THREE.Texture(canvas)`, which results in a `texture` we can use for our sprites. Remember to set `texture.needsUpdate` to `true`, which triggers Three.js to load the actual canvas data into the texture.

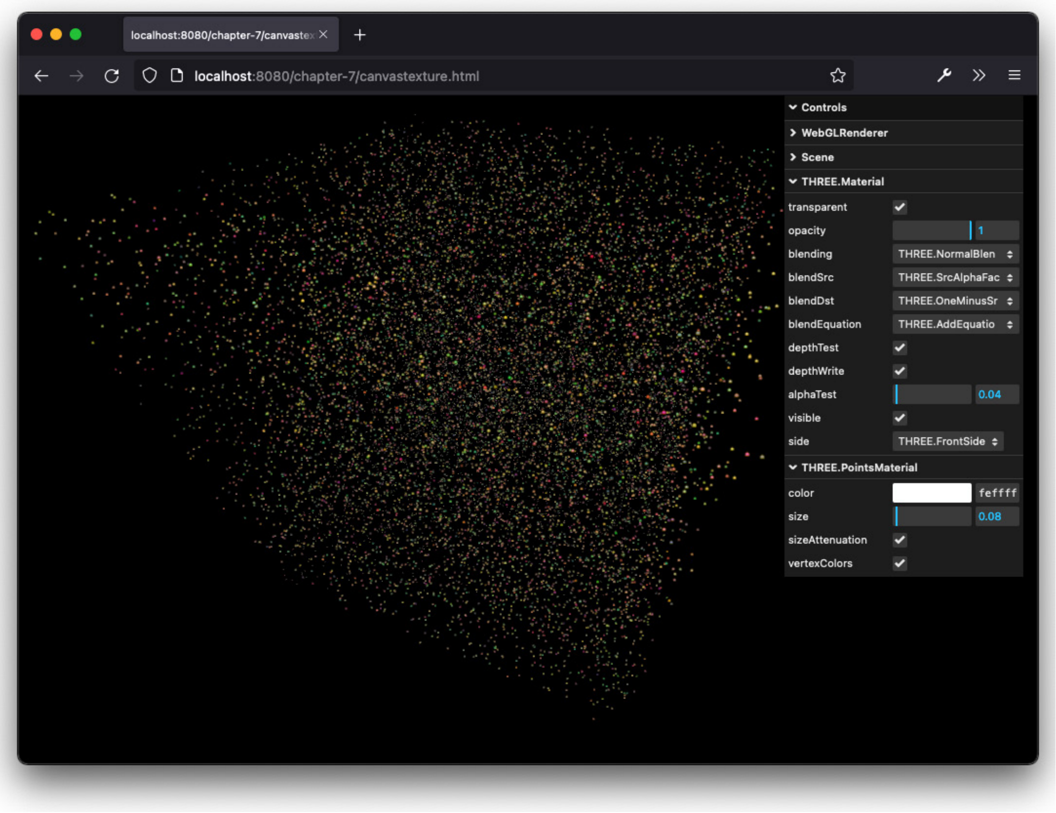
Now that we've got a texture, we can use it to create a `THREE.PointsMaterial`, just like we did in the *Understanding points and sprites* section:

```
const createPoints = () => {
  const points = []
  const range = 15
  for (let i = 0; i < 15000; i++) {
    let particle = new THREE.Vector3(
      Math.random() * range - range / 2,
      Math.random() * range - range / 2,
      Math.random() * range - range / 2
    )
    points.push(particle)
  }
  const colors = new Float32Array(points.length * 3)
  points.forEach((e, i) => {
    const c = new THREE.Color(Math.random() * 0xffffff)
    colors[i * 3] = c.r
    colors[i * 3 + 1] = c.g
    colors[i * 3 + 2] = c.b
  })
  const geom = new THREE.BufferGeometry().setFromPoints(points)
  geom.setAttribute('color', new THREE.BufferAttribute(colors,
3, true))
  return geom
}
const material = new THREE.PointsMaterial({ size: 0.1,
  vertexColors: true, color: 0xffffff, map:
    createGhostTexture() })
```

```
const points = new THREE.Points(createPoint(), material)
```

As you can see, we create `15000` points for this example and position them randomly in the specified range. What you might notice is that even if you turn on `transparency`, some sprites seem to overlap other sprites. This is because Three.js doesn't sort sprites based on their z-index, so during rendering, it can't correctly determine which one is before another one. There are two ways you can work around this: you can turn off `depthWrite`, or you can play around with the `alphaTest` property (starting with 0.5 is a good starting point).

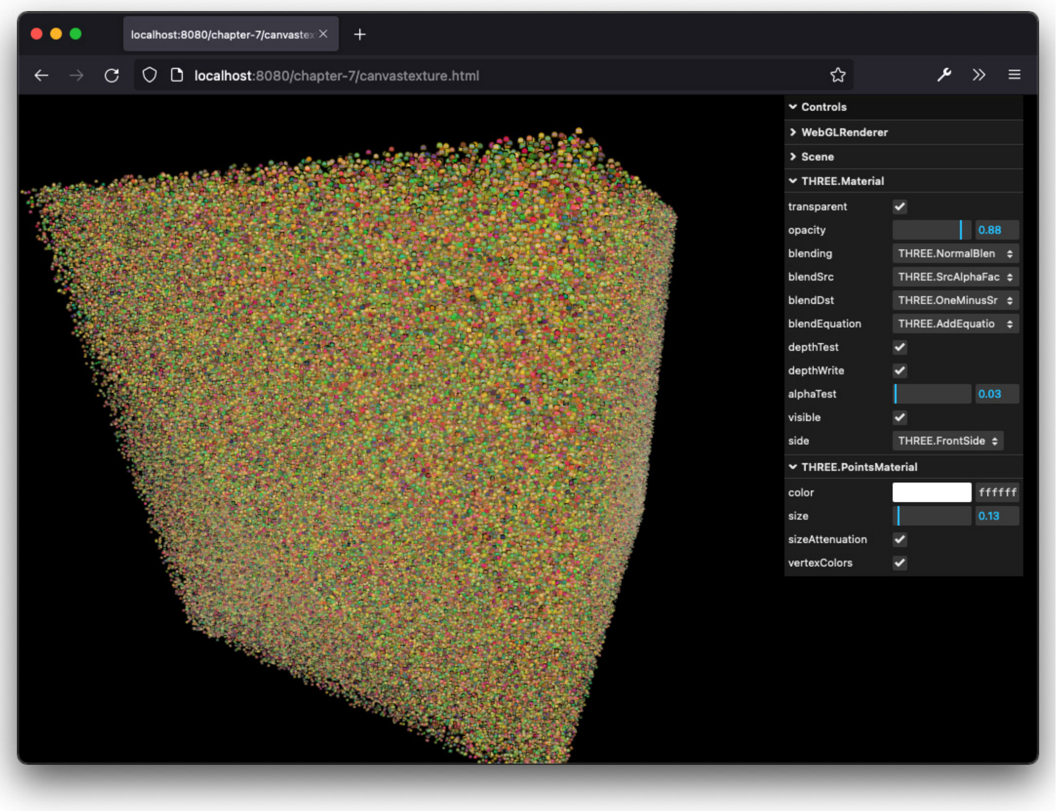
If you zoom out, you will see the 15,000 individual sprites:


localhost:8080/chapter-7/canvastex
localhost:8080/chapter-7/canvastexture.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.PointsMaterial

transparent ✓
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ✓
depthWrite ✓
alphaTest 0.04
visible ✓
side THREE.FrontSide
color feffff
size 0.08
sizeAttenuation ✓
vertexColors ✓

Figure 7.5 – Showing 15,000 sprites all at once

The amazing thing is that even with 1 million points, everything is still rendered very smoothly (of course, this depends on the hardware you're running these examples on):


localhost:8080/chapter-7/canvastex
localhost:8080/chapter-7/canvastexture.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.PointsMaterial

transparent ☑
opacity 0.88
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0.03
visible ☑
side THREE.FrontSide
color ffffff
size 0.13
sizeAttenuation ☑
vertexColors ☑

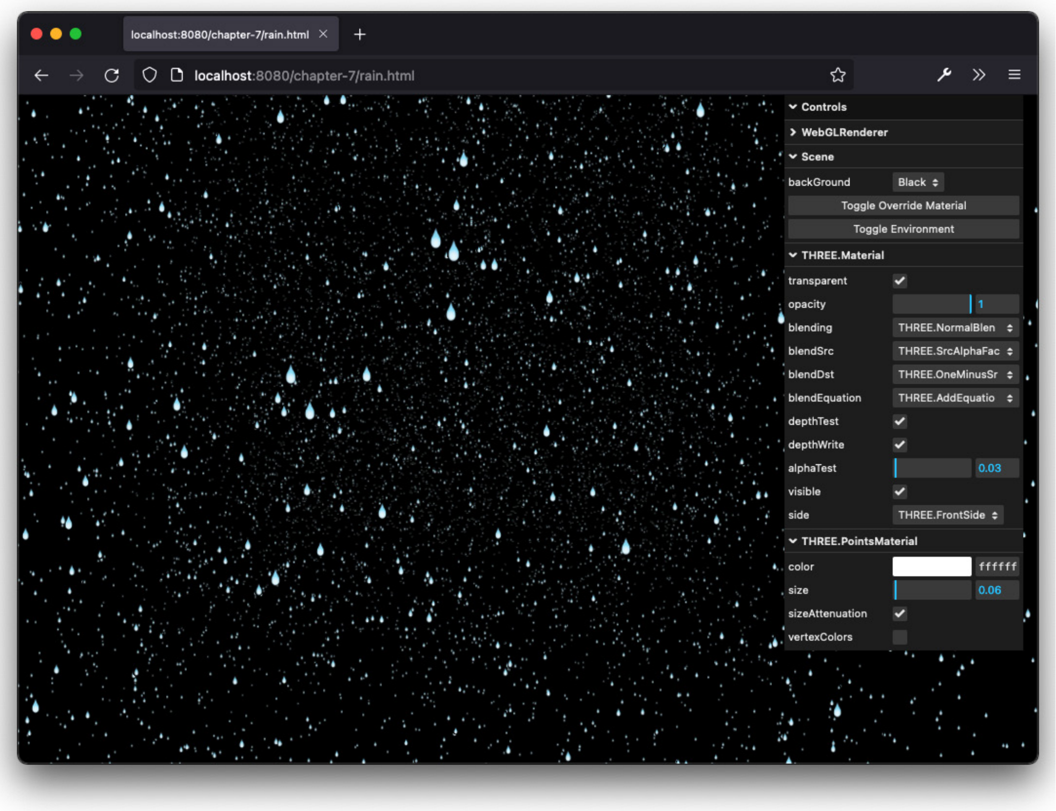
Figure 7.6 – Showing 1 million sprites all at once

In the next section, we'll load some textures from external images and use those instead of drawing the texture ourselves.

Using textures to style particles

In the example shown in the *Drawing an image on the canvas* section, we saw how to style `THREE.Points` using an HTML canvas. Since you can draw anything you want and even load external images, you can use this approach to add all kinds of styles to the particle system. There is, however, a more direct way to use an image to style your particles: you can use the `THREE.TextureLoader().load()` function to load an image as a `THREE.Texture` object. This `THREE.Texture` object can then be assigned to the `map` property of a material.

In this section, we'll show you two examples and explain how to create them. Both these examples use an image as a texture for your particles. In the first example, we'll create a simulation of rain (`rain.html`):


localhost:8080/chapter-7/rain.html
localhost:8080/chapter-7/rain.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
THREE.Material
THREE.PointsMaterial

backGround Black
transparent ☑
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0.03
visible ☑
side THREE.FrontSide
color ffffff
size 0.06
sizeAttenuation ☑
vertexColors ☐

Figure 7.7 – Simulating rain falling down

The first thing we need to do is get a texture that will represent our raindrop. You can find a couple of examples in the `assets/textures/particles` folder. In the upcoming chapters, we will explain all the details and requirements for textures. For now, all you need to know is that the texture should be square and preferably a power of 2 (for example, 64 x 64, 128 x 128, or 256 x 256). For this example, we'll use this texture:

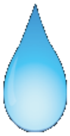


Figure 7.8 – Raindrop texture

This texture is a simple transparent image and shows the shape and color of a raindrop. Before we can use this texture in `THREE.PointsMaterial`, we need to load it. This can be done with the following line of code:

```
const texture = new THREE.TextureLoader().load("../../assets/
textures/particles/raindrop-3t.png");
```

With this line of code, Three.js will load the texture, and we can use it in our material. For this example, we defined the material as follows:

```
const material = new THREE.PointsMaterial({
    size: 0.1,
    vertexColors: false,
    color: 0xffffff,
    map: texture,
    transparent: true,
    opacity: 0.8,
    alphaTest: 0.01
  }),
```

In this chapter, we've discussed all of these properties. The main thing to understand here is that the `map` property points to the texture we loaded with the `THREE.TextureLoader.load` function. Note that we used the `alphaTest` property again to make sure there are no weird artifacts when two sprites are moving in front of one another.

That takes care of styling the `THREE.Points` object. What you'll also see when you open this example is that the points themselves are moving. Doing this is very simple. Each point is represented as a vertex that makes up the geometry that was used to create the `THREE.Points` object. Let's look at how we can add the points for this `THREE.Points` object:

```
const count = 25000
const range = 20
const createPoints = () => {
  const points = []
  for (let i = 0; i < count; i++) {
    let particle = new THREE.Vector3(
      Math.random() * range - range / 2,
      Math.random() * range - range / 2,
      Math.random() * range - range / 1.5
    )
    points.push(particle)
  }
  const velocityArray = new Float32Array(count * 2)
  for (let i = 0; i < count * 2; i += 2) {
    velocityArray[i] = ((Math.random() - 0.5) / 5) * 0.1
    velocityArray[i + 1] = (Math.random() / 5) * 0.1 + 0.01
  }
  const geom = new THREE.BufferGeometry().setFromPoints(points)
  geom.setAttribute('velocity', new THREE.
BufferAttribute(velocityArray, 2))
  return geom
}
const points = new THREE.Points(geom, material);
```

This isn't that different than the previous examples we saw in this chapter. Here, we added another property to each particle called `velocity`. This property consists of two values: `velocityX` and `velocityY`. The first one defines how a particle (a raindrop) moves horizontally, while the second one defines how fast the raindrop falls. Now that each raindrop has its own speed, we can move the individual particles inside the render loop:

```
const positionArray = points.geometry.attributes.position.array
const velocityArray = points.geometry.attributes.velocity.array
for (let i = 0; i < points.geometry.attributes.position.count;
i++) {
  const velocityX = velocityArray[i * 2]
  const velocityY = velocityArray[i * 2 + 1]
  positionArray[i * 3] += velocityX
  positionArray[i * 3 + 1] -= velocityY
  if (positionArray[i * 3] <= -(range / 2) || positionArray[i *
3] >= range / 2)
    positionArray[i * 3] = positionArray[i * 3] * -1
  if (positionArray[i * 3 + 1] <= -(range / 2) ||
positionArray[i * 3 + 1] >= range / 2)
    positionArray[i * 3 + 1] = positionArray[i * 3 + 1] * -1
}
```

```
points.geometry.attributes.position.needsUpdate = true
```

In this piece of code, we get all the vertices (particles) from the geometry that was used to create `THREE.Points`. For each of the particles, we take `velocityX` and `velocityY` and use them to change the current position of the particle. Then, we make sure the particles stay within the range we've defined. If the `v.y` position drops below `0`, we add the raindrop back to the top, and if the `v.x` position reaches any of the edges, we make it bounce back by inverting the horizontal velocity. Finally, we need to tell Three.js we've changed some things in `bufferGeometry` so that it knows the correct values next time we're rendering.

Let's look at another example. This time, we won't make rain; instead, we'll make snow. Additionally, we won't be using just a single texture – we'll use three separate images (taken from the Three.js examples). Let's start by looking at the result first (`snow.html`):

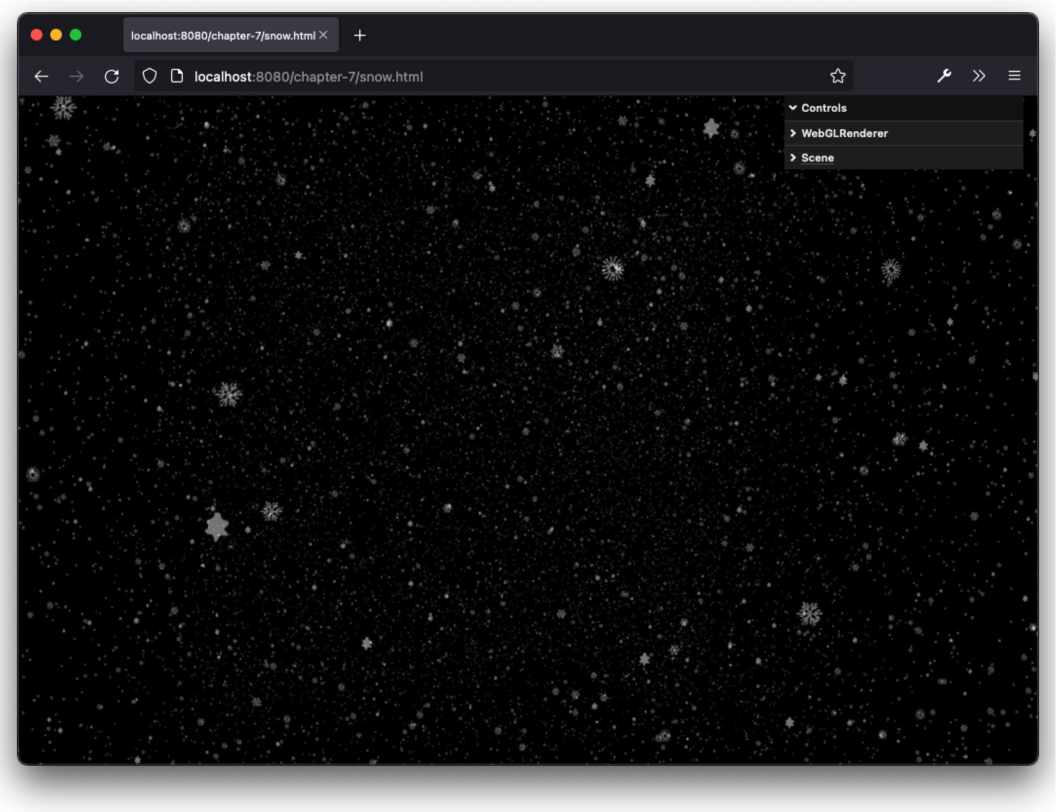

localhost:8080/chapter-7/snow.html
localhost:8080/chapter-7/snow.html
Controls
WebGLRenderer
Scene

Figure 7.9 – Snowy scene based on multiple textures

In the preceding screenshot, if you look closely, you can see that instead of using just a single image as a texture, we've used multiple images that have transparent backgrounds. You might be wondering how we did this. As you probably remember, we can only have a single material for a `THREE.Points` object. If we want to have multiple materials, we just have to make multiple `THREE.Points` instances, as follows:

```
const texture1 = new THREE.TextureLoader().load
  ('/assets/textures/particles/snowflake4_t.png')
const texture2 = new THREE.TextureLoader().load
  ('/assets/textures/particles/snowflake2_t.png')
const texture3 = new  THREE.TextureLoader().load
  ('/assets/textures/particles/snowflake3_t.png')
const baseProps = {
  size: 0.1,
  color: 0xffffff,
  transparent: true,
  opacity: 0.5,
  blending: THREE.AdditiveBlending,
  depthTest: false,
  alphaTest: 0.01
}
const material1 = new THREE.PointsMaterial({
  ...baseProps,
  map: texture1
})
const material2 = new THREE.PointsMaterial({
  ...baseProps,
  map: texture2
})
const material3 = new THREE.PointsMaterial({
  ...baseProps,
  map: texture3
})
const points1 = new THREE.Points(createPoints(), material1)
const points2 = new THREE.Points(createPoints(), material2)
const points3 = new THREE.Points(createPoints(), material3)
```

In this code fragment, you can see that we create three different `THREE.Points` instances, each with its own materials. To move the snowflakes around, we use the same approach as for the rain, so we don't show the details of `createPoint` and the render loop here. One thing to note here is that it is possible to have a single `THREE.Points` instance, where the individual sprites have different textures. However, this would require a custom `fragment-shader` and your own instance of `THREE.ShaderMaterial`.

Before we move on to the next section, note that using `THREE.Points` is a great way to add visual effects to an existing scene. For instance, the snow we saw in the previous example can quickly turn a standard scene into a snowy one:

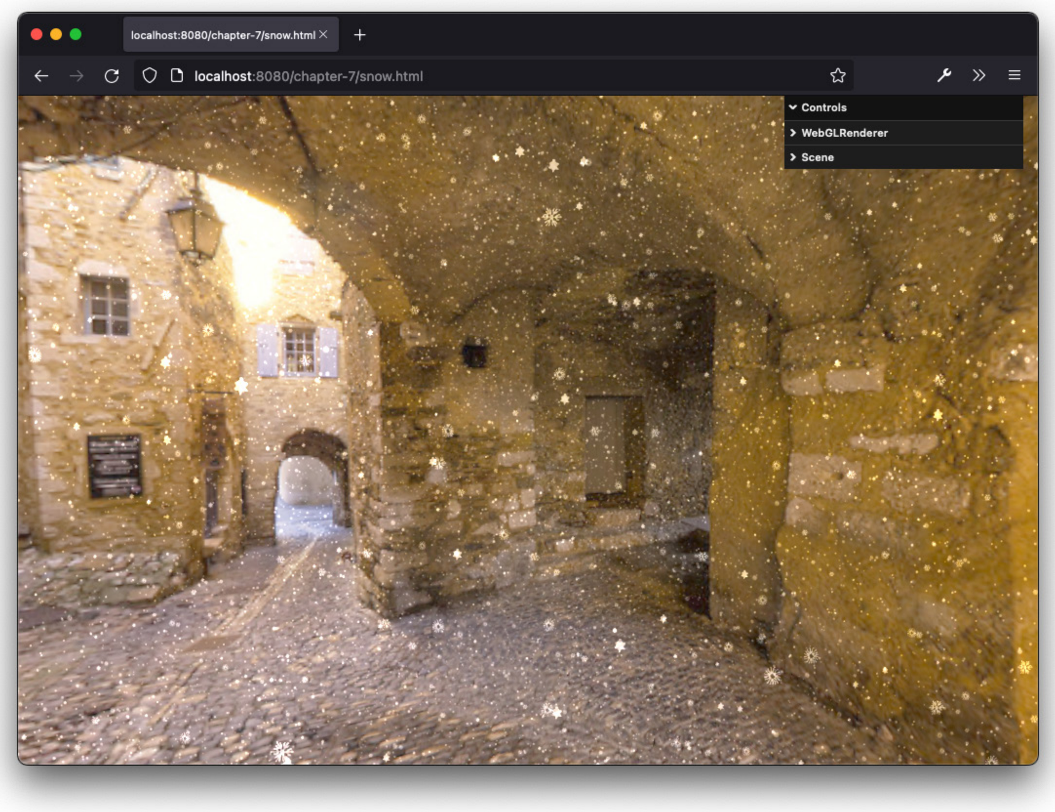

localhost:8080/chapter-7/snow.html
localhost:8080/chapter-7/snow.html
Controls
WebGLRenderer
Scene

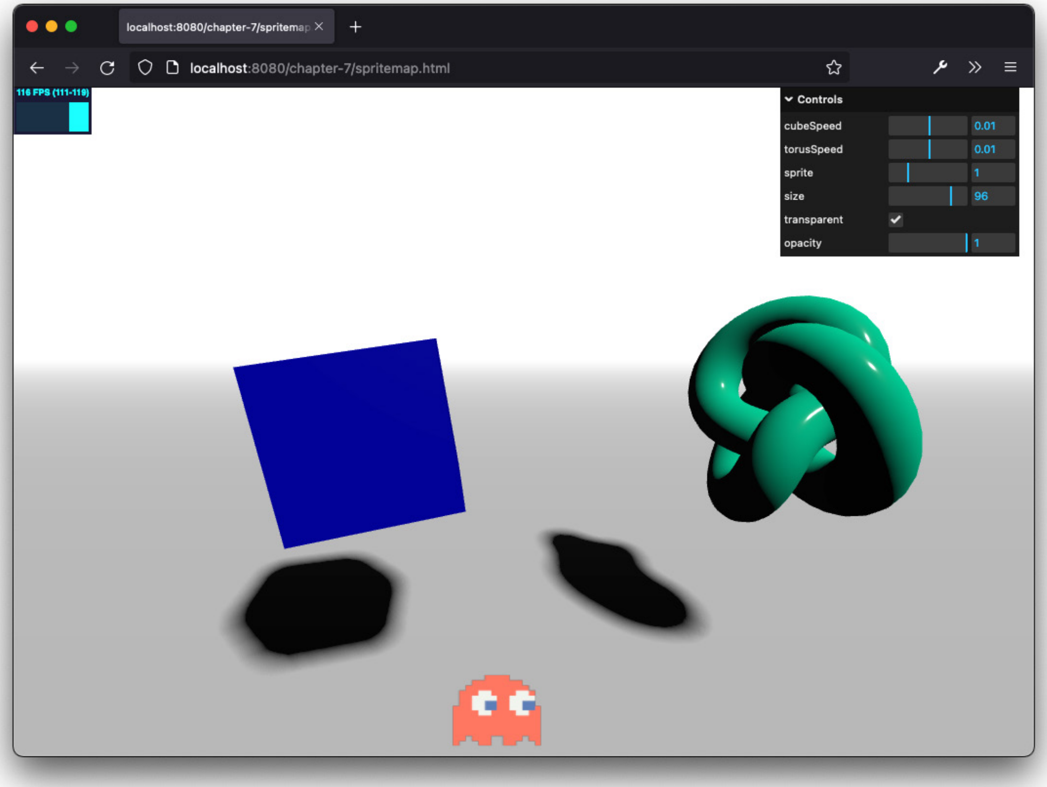
Figure 7.10 – THREE.Points together with a cube map

Another way we can use sprites is to create a simple 2D **heads-up display** (**HUD**) on top of an existing scene. We'll explore how to do this in the next section.

Working with sprite maps

At the beginning of this chapter, we used a `THREE.Sprite` object to render single points. These sprites were positioned somewhere in the 3D world, and their size was based on the distance from the camera (this is also sometimes called **billboarding**). In this section, we'll show an alternative use of the `THREE.Sprite` object: we'll show you how you can use `THREE.Sprite` to create a layer similar to a HUD for your 3D content using an extra `THREE.OrthographicCamera` instance and an additional `THREE.Scene`. We will also show you how to select the image for a `THREE.Sprite` object using a sprite map.

As an example, we're going to create a simple `THREE.Sprite` object that moves from left to right over the screen. In the background, we'll render a 3D scene with a camera, which you can move to illustrate that the `THREE.Sprite` object moves independently of the camera. The following screenshot shows what we'll be creating for the first example (`spritemap.html`):


localhost:8080/chapter-7/spritemap
localhost:8080/chapter-7/spritemap.html
116 FPS (111-119)
Controls

cubeSpeed 0.01
torusSpeed 0.01
sprite 1
size 96
transparent ✓
opacity 1

Figure 7.11 – Using two scenes and cameras to create a HUD

If you open this example in your browser, you'll see a Pac-Man ghost-like sprite moving across the screen that changes color and form whenever it hits the right edge. The first thing we'll do is look at how we can create `THREE.OrthographicCamera` and a separate scene to render this `THREE.Sprite`:

```
const sceneOrtho = new THREE.Scene()
sceneOrtho.backgroundColor = new THREE.Color(0x000000)
const cameraOrtho = new THREE.OrthographicCamera(0, window.
innerWidth, window.innerHeight, 0, -10, 10)
```

Next, let's look at the construction of the `THREE.Sprite` object and how the various shapes the sprite can take are loaded:

```
const getTexture = () => {
  const texture = new THREE.TextureLoader().load
   ('/assets/textures/particles/sprite-sheet.png')
  return texture
}
const createSprite = (size, transparent, opacity, spriteNumber)
=> {
  const spriteMaterial = new THREE.SpriteMaterial({
    opacity: opacity,
    color: 0xffffff,
    transparent: transparent,
    map: getTexture()
  })
  // we have 1 row, with five sprites
  spriteMaterial.map.offset = new THREE.Vector2(0.2 *
spriteNumber, 0)
  spriteMaterial.map.repeat = new THREE.Vector2(1 / 5, 1)
  // make sure the object is always rendered at the front
  spriteMaterial.depthTest = false
  const sprite = new THREE.Sprite(spriteMaterial)
  sprite.scale.set(size, size, size)
  sprite.position.set(100, 50, -10)
  sprite.velocityX = 5
  sprite.name = 'Sprite'
  sceneOrtho.add(sprite)
}
```

In the `getTexture()` function, we load a texture. However, instead of loading five different images for each ghost, we load a single texture that contains all the sprites (also called a sprite map). The image we have as a texture looks like this:

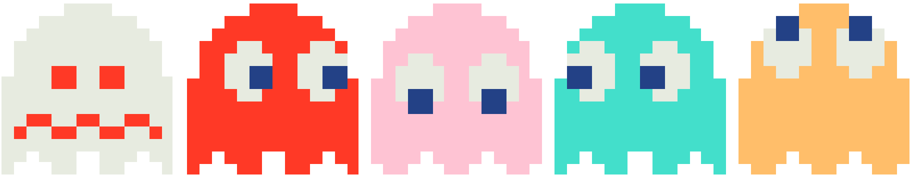


Figure 7.12 – Input sprite sheet

With the `map.offset` and `map.repeat` properties, we can select the correct sprite to show on the screen. With the `map.offset` property, we determine the offset for the *x* axis (`u`) and the *y* axis (`v`) for the texture we loaded. The scale for these properties runs from `0` to `1`. In our example, if we want to select the third ghost, we must set the `u`-offset (*x* axis) to `0.4`, and, because we've only got one row, we don't need to change the `v`-offset (*y* axis). If we only set this property, the texture shows the third, fourth, and fifth ghosts compressed together on screen. To only show one ghost, we need to zoom in. We can do this by setting the `map.repeat` property for the `u`-value to `1/5`. This means that we zoom in (only for the *x* axis) to only show 20% of the texture, which is exactly one ghost.

Finally, we need to update the `render` function:

```
renderer.render(scene, camera)
renderer.autoClear = false
renderer.render(sceneOrtho, cameraOrtho)
```

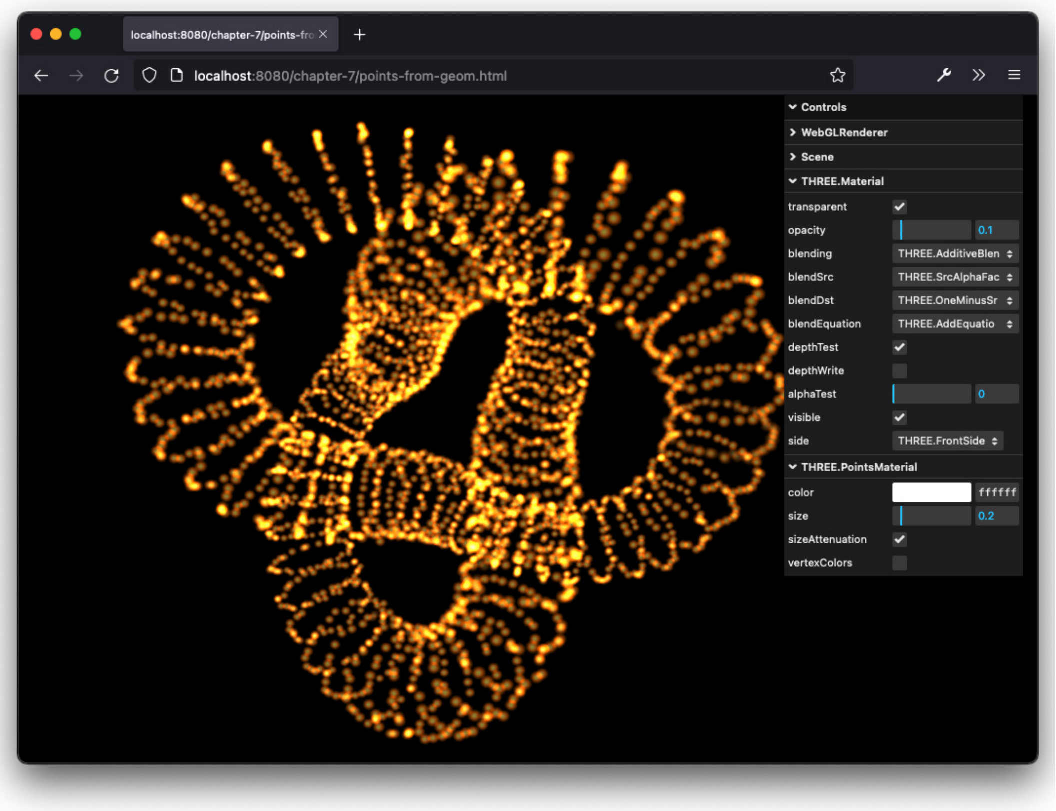
First, we render the scene with the normal camera and the two meshes; after that, we render the scene containing our sprite. In the render loop, we also switch some properties to show the next sprite when it hits the right wall and change the sprite's direction (code not shown).

So far in this chapter, we've mainly looked at creating sprites and point clouds from scratch. An interesting option, though, is to create `THREE.Points` from an existing geometry.

Creating THREE.Points from existing geometry

As you may recall, `THREE.Points` renders each point based on the vertices from the supplied `THREE.BufferGeometry`. This means that if we provide a complex geometry (for example, a torus knot or a tube), we can create `THREE.Points` based on the vertices from that specific geometry. In this final section of this chapter, we'll create a torus knot, like the one we saw in *Chapter 6*, *Exploring Advanced Geometries*, and render it as a `THREE.Points` object.

We explained the torus knot in *Chapter 6*, so we won't go into much detail here. The following screenshot shows the example (`points-from-geom.html`):


localhost:8080/chapter-7/points-fro
localhost:8080/chapter-7/points-from-geom.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.PointsMaterial

transparent ☑
opacity 0.1
blending THREE.AdditiveBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☐
alphaTest 0
visible ☑
side THREE.FrontSide
color ffffff
size 0.2
sizeAttenuation ☑
vertexColors ☐

Figure 7.13 – Torus knot rendered as points with a small animation

As you can see from the preceding screenshot, every vertex used to generate the torus knot is used as a point. We can set this up like this:

```
const texture = new THREE.TextureLoader().load('/assets/
textures/particles/glow.png')
const geometry = new THREE.TorusKnotGeometry(2, 0.5, 100, 30,
2, 3)
const material = new THREE.PointsMaterial({
    size: 0.2,
    vertexColors: false,
    color: 0xffffff,
    map: texture,
    depthWrite: false,
    opacity: 0.1,
    transparent: true,
    blending: THREE.AdditiveBlending
  })
const points = new THREE.Points(geometry, material)
```

As you can see, we simply create a geometry and use that as input for the `THREE.Points` object. This way, we can render every geometry as a points object.

> **Note**
>
> If you load external models using a Three.js model loader (for example, a glTF model), you'll often end up with a hierarchy of objects – often grouped in `THREE.Group` or `THREE.Object3D` objects. In those cases, you'll have to convert each geometry in each group into a `THREE.Points` object.

Summary

That's a wrap for this chapter. We've explained what sprites and points are and how you can style these objects with the materials available. In this chapter, you saw how you can use `THREE.Sprite` directly, and that if you want to create a large number of particles, you should use a `THREE.Points` object. With `THREE.Points`, all the elements share the same material, and the only property you can change for an individual particle is its color by setting the `vertexColors` property of the material to `true` and providing a color value in the `colors` array of `THREE.BufferGeometry`, which is used to create `THREE.Points`. We also showed how you can easily animate particles by changing their position. This works the same for an individual `THREE.Sprite` instance and the vertices from the geometry used to create `THREE.Points` objects.

So far, we have created meshes based on geometries provided by Three.js. This works well for simple models, such as spheres and cubes, but isn't the best approach when you want to create complex 3D models. For those models, you'd usually use a 3D modeling application, such as Blender or 3D Studio Max. In the next chapter, you'll learn how you can load and display models created by such 3D modeling applications.

Part 3:

Particle Clouds, Loading and Animating Models

In this third part, we'll show you how you can load data from external models and how Three.js supports animations. We'll also dive into the different types of textures that are supported by Three.js and how you can use them to enhance your models.

In this part, there are the following chapters:

- *Chapter 8, Creating and Loading Advanced Meshes and Geometries*
- *Chapter 9, Animations and Moving the Camera*
- *Chapter 10, Loading and Working with Textures*

8

Creating and Loading Advanced Meshes and Geometries

In this chapter, we'll look at a couple of different ways that you can create and load advanced and complex geometries and meshes. In *Chapter 5*, *Learning to Work with Geometries*, and *Chapter 6*, *Exploring Advanced Geometries*, we showed you how to create a few advanced geometries using the built-in objects from Three.js. In this chapter, we'll use the following two approaches to create advanced geometries and meshes:

- Geometry grouping and merging
- Loading geometries from external resources

We start with the "group and merge" approach. With this approach, we use the standard Three.js grouping (`THREE.Group`) and the `BufferGeometryUtils.mergeBufferGeometries()` function to create new objects.

Geometry grouping and merging

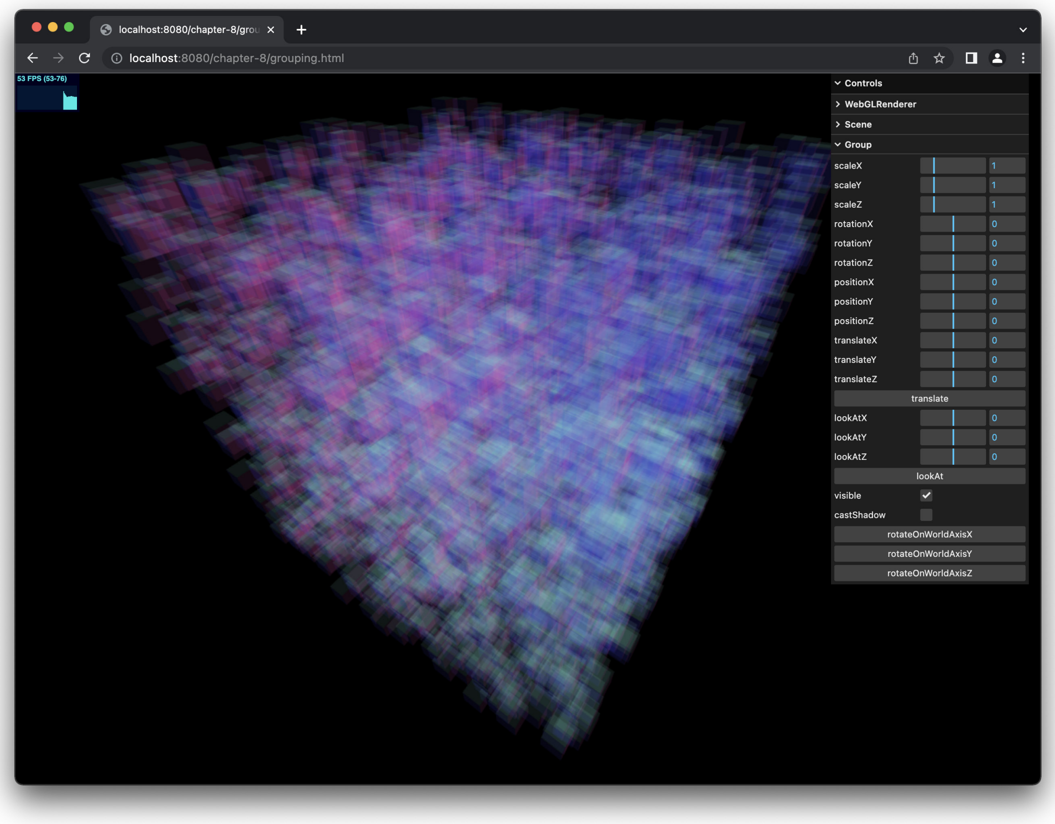
In this section, we'll look at two basic features of Three.js: grouping objects together and merging multiple geometries into a single geometry. We'll start with grouping objects.

Grouping objects together

In some of the previous chapters, you already saw how you can group objects when working with multiple materials. When you create a mesh from a geometry using multiple materials, Three.js creates a group. Multiple copies of your geometry are added to this group, each with its own specific material. This group is returned, so it looks like a mesh that uses multiple materials. In truth, however, it is a group that contains a number of meshes.

Creating groups is very easy. Every mesh you create can contain child elements, which can be added using the `add` function. The effect of adding a child object to a group is that you can move, scale, rotate, and translate the parent object, and all the child objects will also be affected. When using a group, you can still refer to, modify, and position the individual geometries. The only thing you need to remember is that all positions, rotations, and translations are done relative to the parent object.

Let's look at an example (`grouping.html`) in the following screenshot:


localhost:8080/chapter-8/grou
localhost:8080/chapter-8/grouping.html
53 FPS (53-76)
Controls
WebGLRenderer
Scene
Group
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ

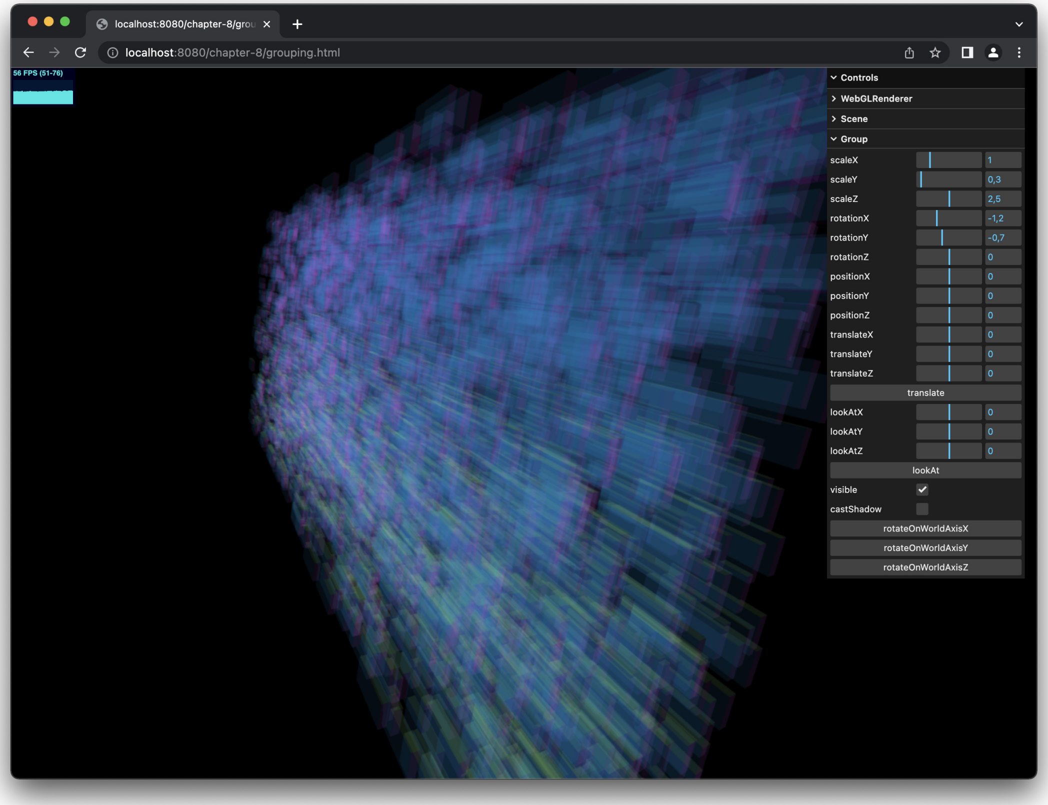
scaleX	1
scaleY	1
scaleZ	1
rotationX	0
rotationY	0
rotationZ	0
positionX	0
positionY	0
positionZ	0
translateX	0
translateY	0
translateZ	0
lookAtX	0
lookAtY	0
lookAtZ	0
visible	☑
castShadow	☐

Figure 8.1 – Using a THREE.Group object to group objects together

In this example, you see a large number of cubes, which are added to the scene as a single group. Before we look at the controls and the effect of using a group, let's have a quick look at how we've created this mesh:

```
const size = 1
const amount = 5000
const range = 20
const group = new THREE.Group()
const mat = new THREE.MeshNormalMaterial()
mat.blending = THREE.NormalBlending
mat.opacity = 0.1
mat.transparent = true
for (let i = 0; i < amount; i++) {
  const x = Math.random() * range - range / 2
  const y = Math.random() * range - range / 2
  const z = Math.random() * range - range / 2
  const g = new THREE.BoxGeometry(size, size, size)
  const m = new THREE.Mesh(g, mat)
  m.position.set(x, y, z)
  group.add(m)
}
```

In this code snippet, you can see that we create a `THREE.Group` instance. This object is almost identical to `THREE.Object3D`, which is the base class of `THREE.Mesh` and `THREE.Scene`, but by itself, it doesn't contain anything or cause anything to be rendered. In this example, we use the `add` function to add a large number of cubes to this scene. For this example, we've added the controls you can use to change the position of a mesh. Whenever you change a property using this menu, the relevant property of the `THREE.Group` object is changed. For instance, in the next example, you can see that when we scale this `THREE.Group` object, all the nested cubes get scaled as well:


localhost:8080/chapter-8/grou
localhost:8080/chapter-8/grouping.html
56 FPS (51-76)
Controls
WebGLRenderer
Scene
Group
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ

scaleX 1
scaleY 0,3
scaleZ 2,5
rotationX -1,2
rotationY -0,7
rotationZ 0
positionX 0
positionY 0
positionZ 0
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑
castShadow ☐

Figure 8.2 – Scaling a group

If you want to experiment a bit more with the `THREE.Group` object, a good exercise would be to alter the example so that the `THREE.Group` instance itself is rotating on the *x* axis while the individual cubes are rotating on their *y* axis.

Performance impact of using THREE.Group

Before we move on to the next section where we look at merging, a quick note on performance. When you use `THREE.Group`, all the individual meshes inside this group are treated as individual objects, which Three.js needs to manage and render. If you've got a large number of objects in the scene, you'll see a noticeable drop in performance. If you look at the top-left corner of *Figure 8.2*, you can see that with 5,000 cubes on screen, we get around 56 **frames per second** (**FPS**). Not too bad, but normally we would run at around 120 FPS.

Three.js provides an additional way where we can still control the individual meshes, but get much better performance. This is done through `THREE.InstancedMesh`. This object works great if you want to render a large number of objects with the same geometry but with different transformations (for example, rotation, scale, color, or any other matrix transformation).

We've created an example called `instanced-mesh.html`, which shows how this works. In this example, we render 250,000 cubes and still have great performance:

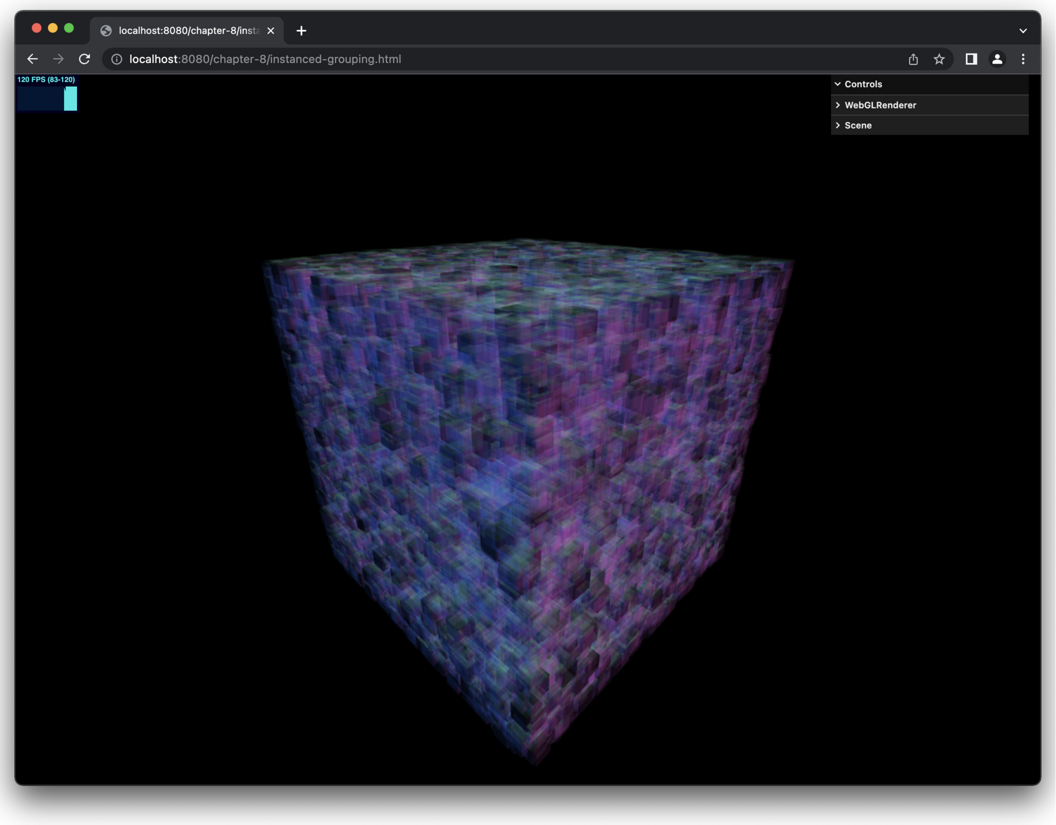

localhost:8080/chapter-8/insta
localhost:8080/chapter-8/instanced-grouping.html
120 FPS (83-120)
Controls
WebGLRenderer
Scene

Figure 8.3 – Using an InstancedMesh object for grouping

To work with a `THREE.InstancedMesh` object, we create it similarly to how we created the `THREE.Group` instance:

```
const size = 1
const amount = 250000
const range = 20
const mat = new THREE.MeshNormalMaterial()
mat.opacity = 0.1
mat.transparent = true
```

```
  mat.blending = THREE.NormalBlending
  const g = new THREE.BoxGeometry(size, size, size)
  const mesh = new THREE.InstancedMesh(g, mat, amount)
  for (let i = 0; i < amount; i++) {
    const x = Math.random() * range - range / 2
    const y = Math.random() * range - range / 2
    const z = Math.random() * range - range / 2
    const matrix = new THREE.Matrix4()
    matrix.makeTranslation(x, y, z)
    mesh.setMatrixAt(i, matrix)
  }
```

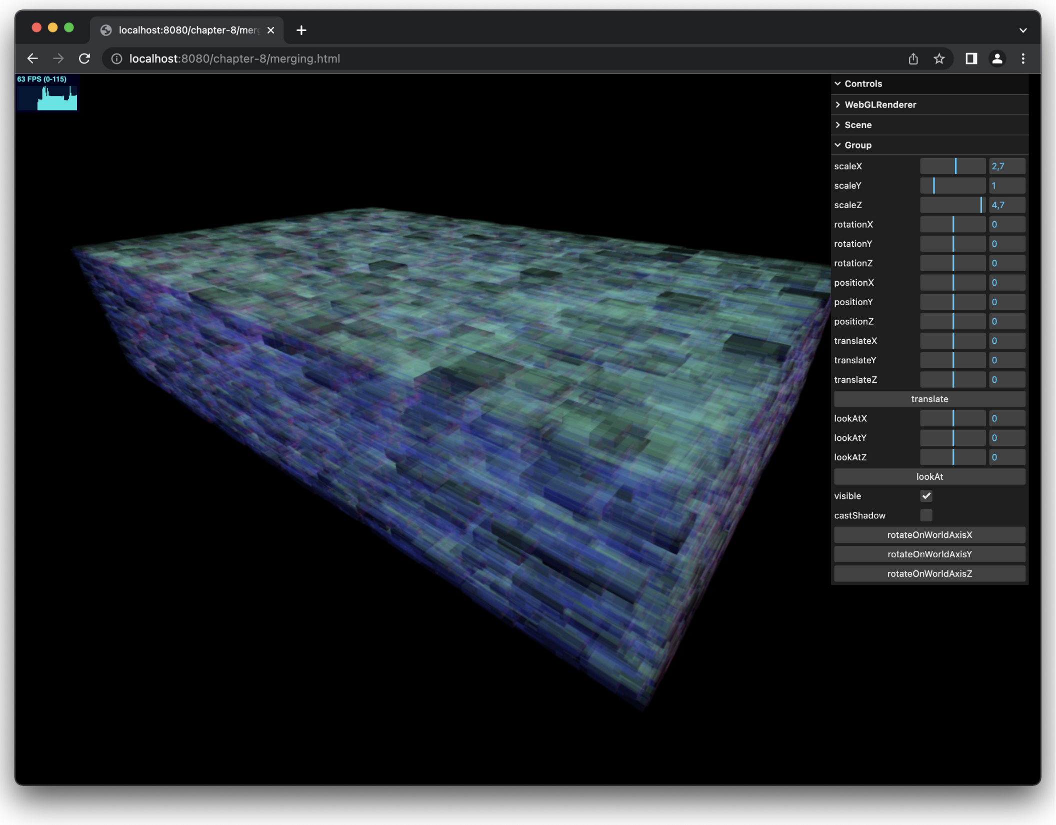
The main difference in creating a `THREE.InstancedMesh` object compared to `THREE.Group` is that we need to define beforehand which material and geometry we want to use and how many instances of this geometry we want to create. To position or rotate one of our instances, we need to provide the transformation using a `THREE.Matrix4` instance. Luckily, we don't need to go into the math behind matrices, since Three.js provides us with a couple of helper functions on the `THREE.Matrix4` instance to define a rotation, a translation, and a couple of other transformations. In this example, we simply position each instance at a random location.

So, if you're working with a small number of meshes (or meshes using different geometries), you should use a `THREE.Group` object if you want to group them together. If you're dealing with a large number of meshes that share a geometry and material, you can use a `THREE.InstancedMesh` object or a `THREE.InstancedBufferGeometry` object for a great performance boost.

In the next section, we'll look at merging, where you'll combine multiple separate geometries and end up with a single `THREE.Geometry` object.

Merging geometries

In most cases, using groups allows you to easily manipulate and manage a large number of meshes. When you're dealing with a very large number of objects, however, performance will become an issue since Three.js has to treat all the children of the group individually. With `BufferGeometryUtils.mergeBufferGeometries`, you can merge geometries together and create a combined one, so Three.js would only have to manage this single geometry. In *Figure 8.4*, you can see how this works and the effect it has on performance. If you open the `merging.html` example, you see a scene again with the same set of randomly distributed semi-transparent cubes, which we merged into a single `THREE.BufferGeometry` object:


localhost:8080/chapter-8/mer
localhost:8080/chapter-8/merging.html
63 FPS (0-115)
Controls
WebGLRenderer
Scene
Group
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ

scaleX 2,7
scaleY 1
scaleZ 4,7
rotationX 0
rotationY 0
rotationZ 0
positionX 0
positionY 0
positionZ 0
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑
castShadow ☐

Figure 8.4 – 500,000 geometries merged into a single geometry

As you can see, we can easily render 50,000 cubes without any drop in performance. To do this, we use the following few lines of code:

```
const size = 1
const amount = 500000
const range = 20
const mat = new THREE.MeshNormalMaterial()
mat.blending = THREE.NormalBlending
mat.opacity = 0.1
mat.transparent = true
const geoms = []
for (let i = 0; i < amount; i++) {
  const x = Math.random() * range - range / 2
```

```
    const y = Math.random() * range - range / 2
    const z = Math.random() * range - range / 2
    const g = new THREE.BoxGeometry(size, size, size)
    g.translate(x, y, z)
    geoms.push(g)
  }
  const merged = BufferGeometryUtils.
    mergeBufferGeometries(geoms)
  const mesh = new THREE.Mesh(merged, mat)
```

In this code snippet, we create a large number of `THREE.BoxGeometry` objects, which we merge together using the `BufferGeometryUtils.mergeBufferGeometries(geoms)` function. The result is a single large geometry, which we can add to the scene. The biggest drawback is that you lose control over the individual cubes since they are all merged into a single large geometry. If you want to move, rotate, or scale a single cube, you can't (unless you search for the correct faces and vertices and position them individually).

Creating new geometries through Constructive Solid Geometry

Besides merging geometries in the way we saw in this chapter, we can also create geometries using **Constructive Solid Geometry** (**CSG**). With CSG, you can apply operations (usually addition, subtraction, difference, and intersection) to combine two geometries. These libraries would create a new geometry, based on the chosen operation. For instance, with CSG, it is very easy to create a solid cube with a sphere-like indentation on one side. Two libraries that you can use for this with Three.js are `three-bvh-csg` (`https://github.com/gkjohnson/three-bvh-csg`) and `Three.csg` (`https://github.com/looeee/threejs-csg`).

With the grouping and merging approach, you can create large and complex geometries using the basic geometries provided by Three.js. If you want to create more advanced geometries, then using the programmatic approach provided by Three.js isn't always the best and easiest option. Three.js, luckily, offers a couple of other options to create geometries. In the next section, we'll look at how you can load geometries and meshes from external resources.

Loading geometries from external resources

Three.js can read a large number of 3D file formats and import geometries and meshes defined in those files. A note here is that not all the features of these formats are always supported. So, sometimes there might be an issue with the textures, or materials might not be set up correctly. The new de facto standard for exchanging models and textures is **glTF**, so if you want to load externally created models, exporting those models to glTF format will usually give you the best results in Three.js.

In this section, we'll dive a bit deeper into some of the formats that are supported by Three.js, but we won't show you all the loaders. The following list shows an overview of the formats supported by Three.js:

- **AMF**: AMF is another 3D printing standard, but isn't under active development anymore. The following *Wikipedia* page has additional information on this standard: `https://www.sculpteo.com/en/glossary/amf-definition/`.
- **3DM**: 3DM is the format used by Rhinoceros, which is a tool to create 3D models. More information on Rhinoceros can be found here: `https://www.rhino3d.com/`.
- **3MF**: 3MF is one of the standards used in 3D printing. Information about this format can be found on the *3MF Consortium* home page: `https://3mf.io`.
- **COLLAborative Design Activity (COLLADA)**: COLLADA is a format for defining digital assets in an XML-based format. This is a widely used format that is supported by pretty much all 3D applications and rendering engines.
- **Draco**: Draco is a file format for storing geometries and point clouds in a very efficient way. It specifies how these elements are best compressed and decompressed. Details about how Draco works can be found on its GitHub page: `https://github.com/google/draco`.
- **GCode**: GCode is a standard way of talking to 3D printers or **CNC** machines. When a model is printed, one of the ways a 3D printer can be controlled is by sending it GCode commands. The details of this standard are described in the following paper: `https://www.nist.gov/publications/nist-rs274ngc-interpreter-version-3?pub_id=823374`.
- **glTF**: This is a specification that defines how 3D scenes and models can be exchanged and loaded by different applications and tools and is becoming the standard format for exchanging models on the web. They come in a binary format with the `.glb` extension and a text-based format with the `.gltf` extension. More information on this standard can be found here: `https://www.khronos.org/gltf/`.
- **Industry Foundation Classes (IFC)**: This is an open file format used by **building information modeling** (**BIM**) tools. It contains a model of a building and a lot of additional information on the materials used. More information about this standard can be found here: `https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes/`.
- **JSON**: Three.js has its own JSON format that you can use to declaratively define a geometry or a scene. Even though this isn't an official format, it's very easy to use and comes in very handy when you want to reuse complex geometries or scenes.
- **KMZ**: This is the format used for 3D assets on Google Earth. More information can be found here: `https://developers.google.com/kml/documentation/kmzarchives`.
- **LDraw**: LDraw is an open standard you can use to create virtual LEGO models and scenes. More information can be found on the LDraw home page: `https://ldraw.org`.

- **LWO**: This is the file format used by LightWave 3D. More information on LightWave 3D can be found here: `https://www.lightwave3d.com/`.
- **NRRD**: NRRD is a file format used to visualize volumetric data. It can, for instance, be used to render CT scans. A lot of information and samples can be found here: `http://teem.sourceforge.net/nrrd/`.
- **OBJ** and **MTL**: OBJ is a simple 3D format first developed by Wavefront Technologies. It's one of the most widely adopted 3D file formats and is used to define the geometry of an object. MTL is a companion format to OBJ. In an MTL file, the material of the objects in an OBJ file is specified. Three.js also has a custom OBJ exporter, called `OBJExporter`, should you want to export your models to OBJ from Three.js.
- **PCD**: This is an open format for describing point clouds. More information can be found here: `https://pointclouds.org/documentation/tutorials/pcd_file_format.html`.
- **PDB**: This is a very specialized format, created by **Protein Data Bank** (**PDB**), which is used to specify what proteins look like. Three.js can load and visualize proteins specified in this format.
- **Polygon File Format (PLY)**: This is most often used to store information from 3D scanners.
- **Packed Raw WebGL Model (PRWM)**: This is another format focusing on the efficient storage and parsing of 3D geometries. More information on this standard and how you can use it is described here: `https://github.com/kchapelier/PRWM`.
- **STereoLithography (STL)**: This is widely used for rapid prototyping. For instance, models for 3D printers are often defined as STL files. Three.js also has a custom STL exporter, called `STLExporter.js`, should you want to export your models to STL from Three.js.
- **SVG**: SVG is a standard way to define vector graphics. This loader allows you to load an SVG file and returns a set of `THREE.Path` elements that you can use for extruding or rendering in 2D.
- **3DS**: The Autodesk 3DS format. More information can be found at `https://www.autodesk.com/`.
- **TILT**: TILT is the format used by Tilt Brush, a VR tool that allows you to paint in VR. More information is available here: `https://www.tiltbrush.com/`.

- **VOX**: The format used by MagicaVoxel, a free tool you can use to create voxel art. More information is available on the home page of MagicaVoxel: `https://ephtracy.github.io/`.
- **Virtual Reality Modeling Language (VRML)**: This is a text-based format that allows you to specify 3D objects and worlds. It has been superseded by the X3D file format. Three.js doesn't support loading **X3D** models, but these models can be easily converted to other formats. More information can be found at `http://www.x3dom.org/?page_id=532#`.
- **Visualization Toolkit (VTK)**: This is the file format defined by and used to specify vertices and faces. There are two formats available: a binary one and a text-based **ASCII** one. Three.js only supports the ASCII-based format.
- **XYZ**: This is a very simple file format for describing points in 3D space. More information is available here: `https://people.math.sc.edu/Burkardt/data/xyz/xyz.html`.

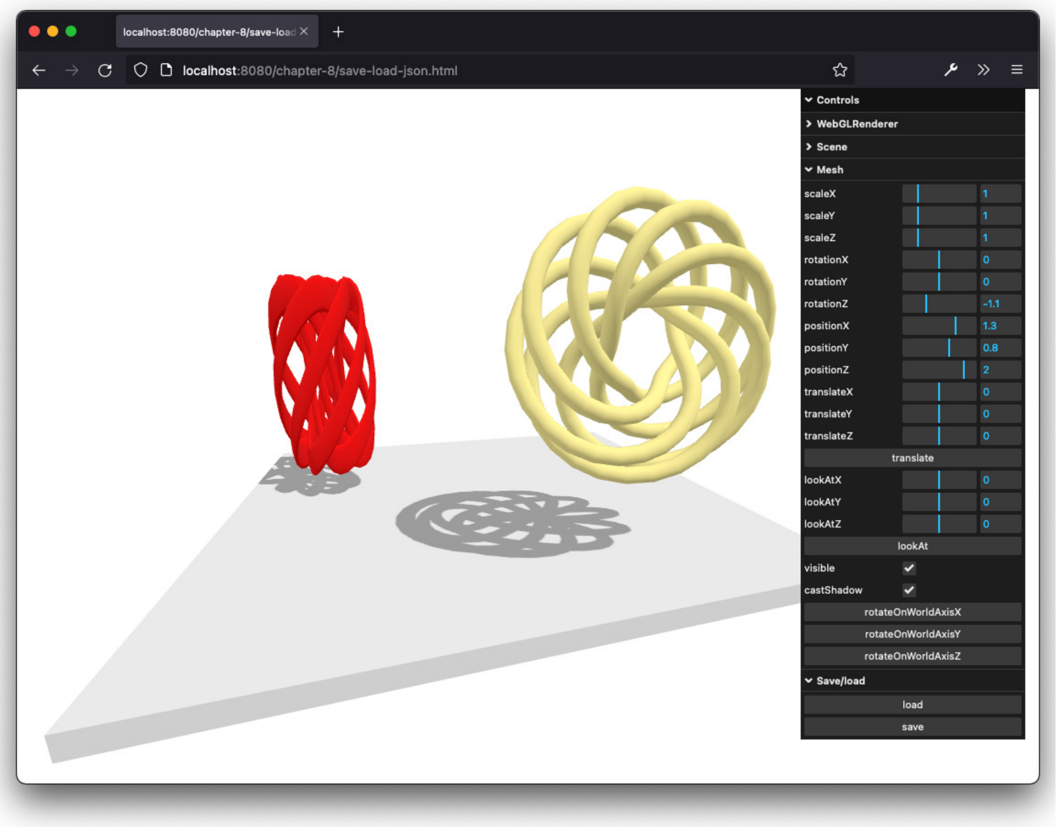
In *Chapter 9*, *Animations and Moving the Camera*, we'll revisit some of these formats (and look at a number of additional ones) when we look at animations.

As you can see from this list, Three.js supports a very large number of 3D file formats. We won't be describing all of them, just the most interesting ones. We'll start with the JSON loader since that provides a nice way to store and retrieve scenes you've created yourself.

Saving and loading in Three.js JSON format

You can use the Three.js JSON format for two different scenarios in Three.js. You can use it to save and load a single `THREE.Object3D` object (which means you can also use it to export a `THREE.Scene` object).

To demonstrate saving and loading, we created a simple example based on `THREE.TorusKnotGeometry`. With this example, you can create a torus knot, just as we did in *Chapter 5*, and, using the **save** button from the **Save/load** menu, you can save the current geometry. For this example, we save using the HTML5 local storage API. This API allows us to easily store persistent information in the client's browser and retrieve it at a later time (even after the browser has been shut down and restarted):


localhost:8080/chapter-8/save-load
localhost:8080/chapter-8/save-load-json.html
Controls
WebGLRenderer
Scene
Mesh
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ
Save/load
load
save

scaleX 1
scaleY 1
scaleZ 1
rotationX 0
rotationY 0
rotationZ -1.1
positionX 1.3
positionY 0.8
positionZ 2
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑
castShadow ☑

Figure 8.5 – Showing the loaded and the current mesh

In the previous screenshot, you can see two meshes—the red one is the one we loaded, and the yellow one is the original one. If you open this example yourself and click the **save** button, the current state of the mesh will be stored. Now, you can refresh the browser and click **load**, and the saved state will be shown in red.

Exporting in JSON from Three.js is very easy and doesn't require you to include any additional libraries. The only thing you need to do is to export `THREE.Mesh` as JSON and store it in the browser's `localstorage`, as follows:

```
const asJson = mesh.toJSON()
localStorage.setItem('json', JSON.stringify(asJson))
```

Before saving it, we first convert the result from the `toJSON` function (a JavaScript object) to a string using the `JSON.stringify` function. To save this information using the HTML5 local storage

API, all we have to do is call the `localStorage.setItem` function. The first argument is the key value (json) that we can later use to retrieve the information we passed in as the second argument.

This JSON string looks like this:

```
{
  "metadata": {
    "version": 4.5,
    "type": "Object",
    "generator": "Object3D.toJSON"
  },
  "geometries": [
    {
      "uuid": "15a98944-91a8-45e0-b974-0d505fcd12a8",
      "type": "TorusKnotGeometry",
      "radius": 1,
      "tube": 0.1,
      "tubularSegments": 200,
      "radialSegments": 10,
      "p": 6,
      "q": 7
    }
  ],
  "materials": [
    {
      "uuid": "38e11bca-36f1-4b91-b3a5-0b2104c58029",
      "type": "MeshStandardMaterial",
      "color": 16770655,
      // left out some material properties
      "stencilFuncMask": 255,
      "stencilFail": 7680,
      "stencilZFail": 7680,
      "stencilZPass": 7680
    }
  ],
  "object": {
    "uuid": "373db2c3-496d-461d-9e7e-48f4d58a507d",
    "type": "Mesh",
    "castShadow": true,
    "layers": 1,
    "matrix": [
      0.5,
      ...
      1
    ],
    "geometry": "15a98944-91a8-45e0-b974-0d505fcd12a8",
    "material": "38e11bca-36f1-4b91-b3a5-0b2104c58029"
  }
}
```

As you can see, Three.js saves all the information about the `THREE.Mesh` object. Loading `THREE.Mesh` back into Three.js also requires just a few lines of code, as follows:

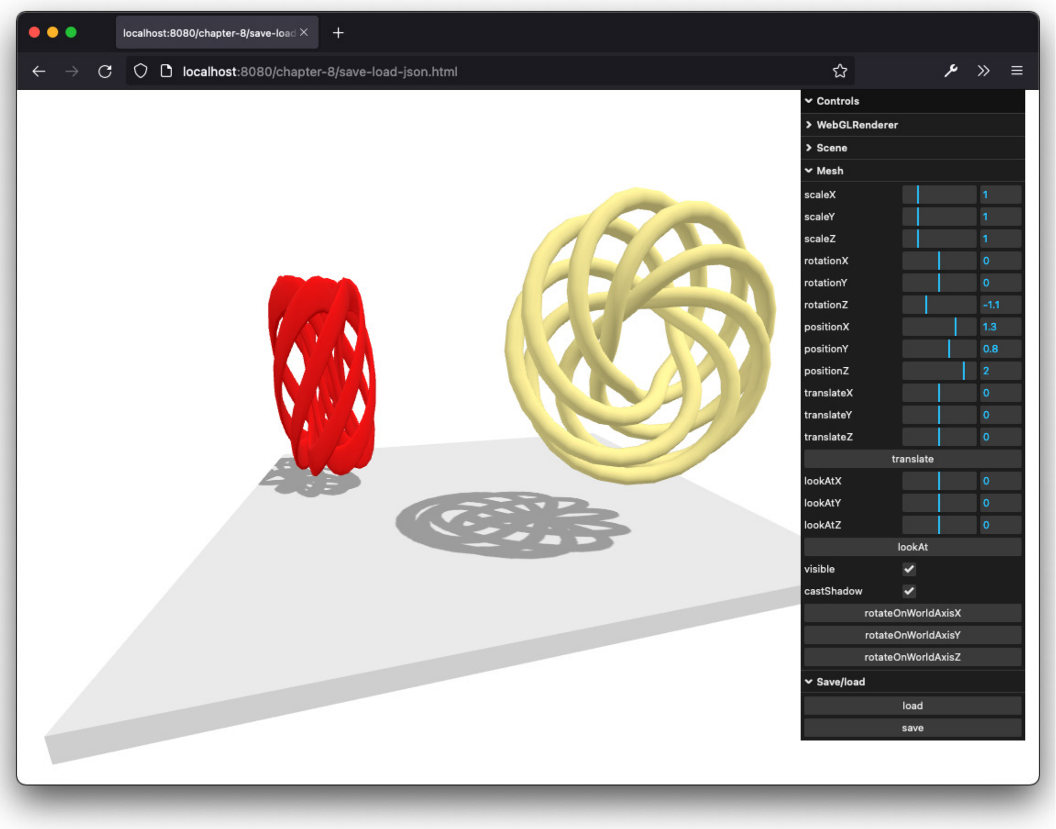
```
const fromStorage = localStorage.getItem('json')
if (fromStorage) {
  const structure = JSON.parse(fromStorage)
  const loader = new THREE.ObjectLoader()
  const mesh = loader.parse(structure)
  mesh.material.color = new THREE.Color(0xff0000)
  scene.add(mesh)
}
```

Here, we first get the JSON from local storage using the name we saved it with (json, in this case). For this, we use the `localStorage.getItem` function provided by the HTML5 local storage API. Next, we need to convert the string back to a JavaScript object (`JSON.parse`) and convert the JSON object back to `THREE.Mesh`. Three.js provides a helper object called `THREE.ObjectLoader`, which you can use to convert JSON to `THREE.Mesh`. In this example, we used the `parse` method on the loader to directly parse a JSON string. The loader also provides a load function, where you can pass the URL to a file containing the JSON definition.

As you can see here, we only saved a `THREE.Mesh` object, so we lose everything else. If you want to save the complete scene, including the lights and the cameras, you can use the same approach to export a scene:

```
const asJson = scene.toJSON()
localStorage.setItem('scene', JSON.stringify(asJson))
```

The result of this is a complete scene description in JSON:



localhost:8080/chapter-8/save-load-json.html

scaleX 1
scaleY 1
scaleZ 1
rotationX 0
rotationY 0
rotationZ -1.1
positionX 1.3
positionY 0.8
positionZ 2
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑
castShadow ☑

localhost:8080/chapter-8/save-load
Controls
WebGLRenderer
Scene
Mesh
translate
lookAt
rotateOnWorldAxisX
rotateOnWorldAxisY
rotateOnWorldAxisZ
Save/load
load
save

Figure 8.6 – Exporting a scene to JSON

This can be loaded in the same way as we already showed for a `THREE.Mesh` object. While storing your current scene and objects in JSON comes in very handy when you're working exclusively in Three.js, this isn't a format that can easily be exchanged with or created by other tools and programs. In the next section, we'll look a bit deeper into some of the 3D formats supported by Three.js.

Importing from 3D file formats

At the beginning of this chapter, we listed a number of formats that are supported by Three.js. In this section, we'll quickly walk through a few examples of those formats.

The OBJ and MTL formats

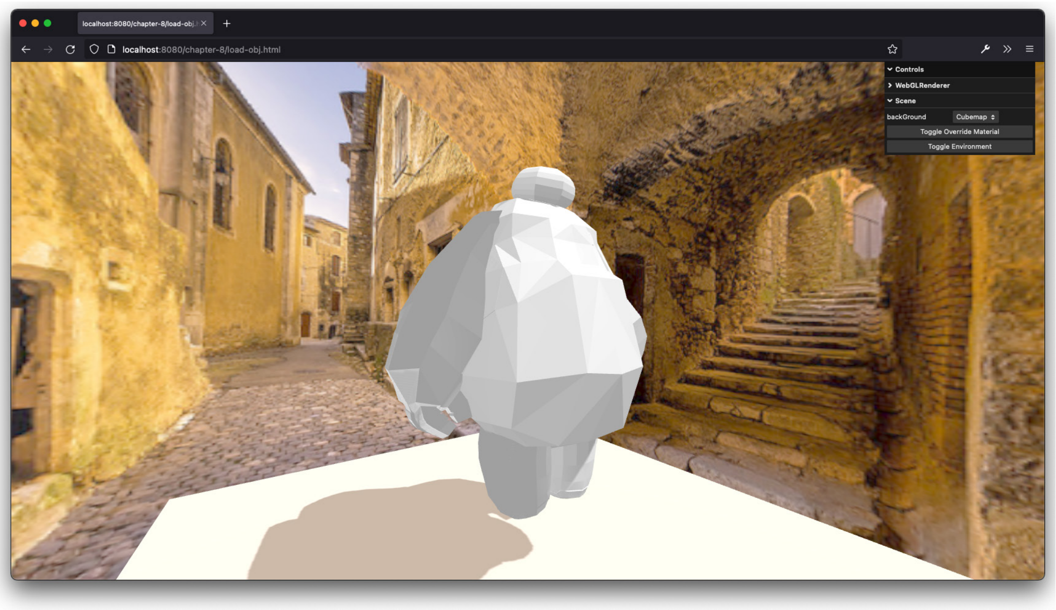
OBJ and MTL are companion formats and are often used together. An OBJ file defines the geometry, and an MTL file defines the materials that are used. Both OBJ and MTL are text-based formats. A part of an OBJ file looks like this:

```
v -0.032442 0.010796    0.025935
v -0.028519 0.013697    0.026201
v -0.029086 0.014533    0.021409
usemtl Material
s   1
f   2731    2735 2736 2732
f   2732    2736 3043 3044
```

An MTL file defines materials, as follows:

```
newmtl Material
Ns  56.862745
Ka  0.000000    0.000000    0.000000
Kd  0.360725    0.227524    0.127497
Ks  0.010000    0.010000    0.010000
Ni  1.000000
d 1.000000
illum 2
```

The OBJ and MTL formats are well supported by Three.js, so this is a good format to choose if you want to exchange 3D models. Three.js has two different loaders you can use. If you only want to load the geometry, you use `OBJLoader`. We used this loader for our example (`load-obj.html`). The following screenshot shows this example:



localhost:8080/chapter-8/load-obj.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment

backGround Cubemap

Figure 8.7 – OBJ model that just defines the geometry

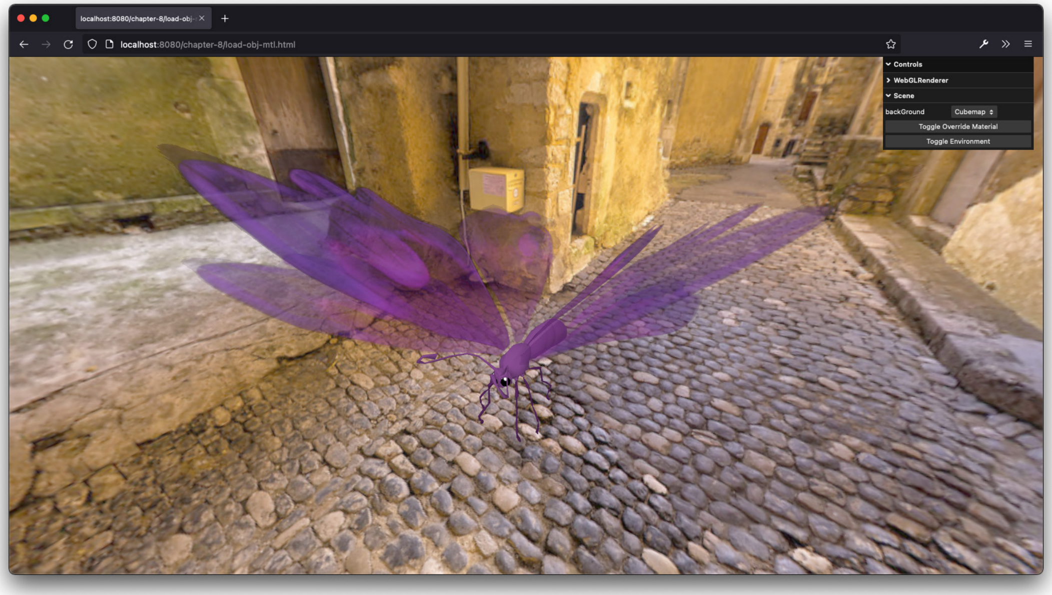
Loading an OBJ model from an external file is done like this:

```
import { OBJLoader } from 'three/examples/jsm/
  loaders/OBJLoader'
new OBJLoader().loadAsync('/assets/models/
  baymax/Bigmax_White_OBJ.obj').then((model) => {
  model.scale.set(0.05, 0.05, 0.05)
  model.translateY(-1)
  visitChildren(model, (child) => {
    child.receiveShadow = true
    child.castShadow = true
  })
  return model
})
```

In this code, we use `OBJLoader` to load the model from a URL asynchronously. This returns a JavaScript promise, which, when resolved, will contain the mesh. Once the model is loaded, we do some fine-tuning and make sure the model casts shadows and receives shadows as well. Besides `loadAsync`, each loader also provides a `load` function, which, instead of working with promises, works with callbacks. This same code would then look something like this:

```
const model = new OBJLoader().load('/assets/models/baymax
  /Bigmax_White_OBJ.obj', (model) => {
  model.scale.set(0.05, 0.05, 0.05)
  model.translateY(-1)
  visitChildren(model, (child) => {
    child.receiveShadow = true
    child.castShadow = true
  })
  // do something with the model
  scene.add(model)
})
```

In this chapter, we'll use the `Promise`-based `loadAsync` approach, since that avoids having nested callbacks and makes it a bit easier to chain these kinds of calls together. The next example (`oad-obj-mtl.html`) uses `OBJLoader`, together with `MTLLoader`, to load a model and directly assign a material. The following screenshot shows this example:


localhost:8080/chapter-8/load-obj-
localhost:8080/chapter-8/load-obj-mtl.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment

backGround Cubemap

Figure 8.8 – OBJ.MTL model with a model and materials

Using an `MTL` file besides the `OBJ` file follows the same principle we saw earlier in this section:

```
const model = mtlLoader.loadAsync('/assets/models/butterfly/
  butterfly.mtl').then((materials) => {
  objLoader.setMaterials(materials)
  return objLoader.loadAsync('/assets/models/butterfly/
    butterfly.obj').then((model) => {
    model.scale.set(30, 30, 30)
    visitChildren(model, (child) => {
      // if there are already normals, we can't merge
        vertices
      child.geometry.deleteAttribute('normal')
      child.geometry = BufferGeometryUtils.
        mergeVertices(child.geometry)
      child.geometry.computeVertexNormals()
      child.material.opacity = 0.1
      child.castShadow = true
    })
    const wing1 = model.children[4]
    const wing2 = model.children[5]
    [0, 2, 4, 6].forEach(function (i) {
      model.children[i].rotation.z = 0.3 * Math.PI })
    [1, 3, 5, 7].forEach(function (i) {
      model.children[i].rotation.z = -0.3 * Math.PI })
    wing1.material.opacity = 0.9
    wing1.material.transparent = true
    wing1.material.alphaTest = 0.1
    wing1.material.side = THREE.DoubleSide
    wing2.material.opacity = 0.9
    wing2.material.depthTest = false
    wing2.material.transparent = true
    wing2.material.alphaTest = 0.1
    wing2.material.side = THREE.DoubleSide
    return model
  })
})
```

The first thing to mention before we look at the code is that if you receive an `OBJ` file, an `MTL` file, and the required texture files, you'll have to check how the `MTL` file references the textures. These should be referenced relative to the `MTL` file and not as an absolute path. The code itself isn't that different than the one we saw for `THREE.ObjLoader`. The first thing we do is load the `MTL` file with a `THREE.MTLLoader` object and the loaded materials are set in `THREE.ObjLoader` through the `setMaterials` function.

The model we've used as an example, in this case, is complex. So, we set some specific properties in the callback to fix a number of rendering issues, as follows:

- We needed to merge the vertices in the model so that it is rendered as a smooth model. For this, we first needed to remove the already defined `normal` vectors from the loaded model so that we could use the `BufferGeometryUtils.mergeVertices` and `computeVertexNormals` functions to provide Three.js with the information to correctly render the model.
- The opacity in the source files was set incorrectly, which caused the wings to be invisible. So, to fix that, we set the `opacity` and `transparent` properties ourselves.
- By default, Three.js only renders one side of an object. Since we look at the wings from two sides, we needed to set the `side` property to the `THREE.DoubleSide` value.
- The wings caused some unwanted artifacts when they needed to be rendered on top of one another. We fixed that by setting the `alphaTest` property.

But as you can see, you can easily load complex models directly into Three.js and render them in real time in your browser. You might need to fine-tune various material properties, though.

Loading a gLTF model

We already mentioned that glTF is a great format to use when importing data in Three.js. Just to show you how easy it is to import and show even complex scenes, we've added an example where we just took a model from `https://sketchfab.com/3d-models/sea-house-bc4782005e9646fb9e6e18df61bfd28d`:

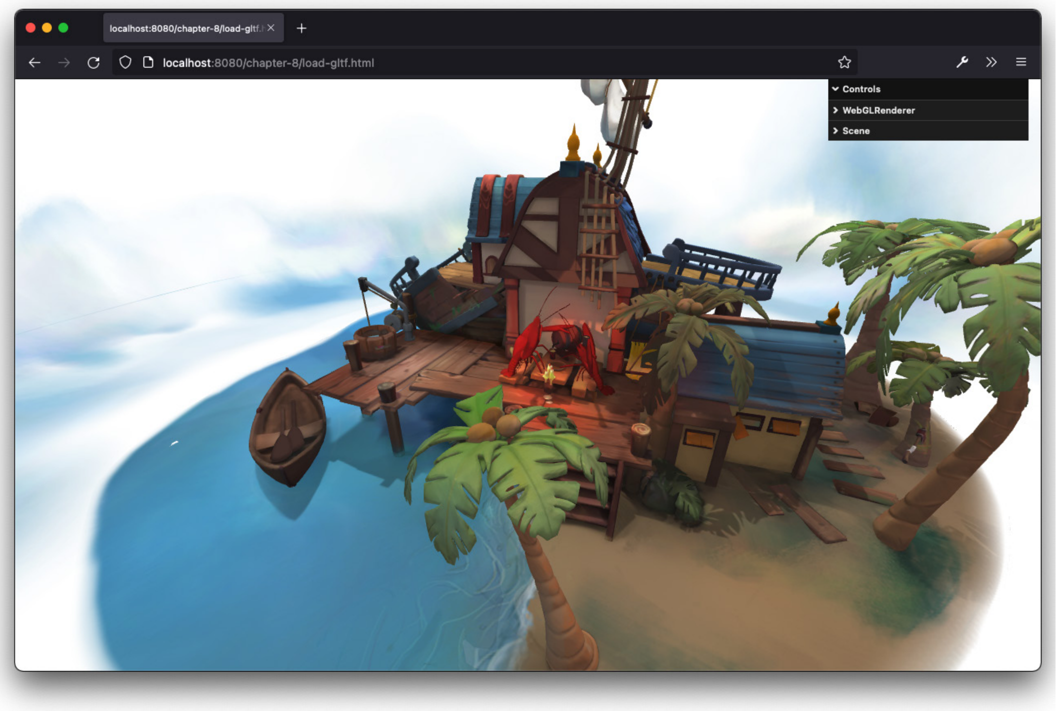

localhost:8080/chapter-8/load-gltf.h
localhost:8080/chapter-8/load-gltf.html
Controls
WebGLRenderer
Scene

Figure 8.9 – Complex 3D scene loaded from glTF with Three.js

As you can see from the previous screenshot, this isn't a simple scene but a complex one, with lots of models, textures, shadows, and other elements. To get this in Three.js, all we had to do was this:

```
const loader = new GLTFLoader()
return loader.loadAsync('/assets/models/sea_house/
  scene.gltf').then((structure) => {
  structure.scene.scale.setScalar(0.2, 0.2, 0.2)
  visitChildren(structure.scene, (child) => {
    if (child.material) {
      child.material.depthWrite = true
    }
  })
  scene.add(structure.scene)
})
```

You're already familiar with the async loader, and the only thing we needed to fix was to make sure the `depthWrite` property of the materials was set correctly (this seems to be a common issue with some glTF models). And that's it—it just works. glTF also allows us to define animations, which is something we'll look at a bit closer in the next chapter.

Showing complete LEGO models

Besides 3D models, where the model defines the vertices, materials, lights, and more, there are also various file formats that don't explicitly define the geometries but have more specific usage. The `LDrawLoader` loader, which we'll be looking at in this section, was created to render LEGO models in 3D. Using this loader works in the same way as we've already seen a couple of times:

```
loader.loadAsync('/assets/models/lego/10174-1-ImperialAT-ST-
UCS.mpd_Packed.mpd').'/assets/models/lego/10174-1-ImperialAT-
ST-UCS.mpd_Packed.mpd'.then((model) => {
  model.scale.set(0.015, 0.015, 0.015)
  model.rotateZ(Math.PI)
  model.rotateY(Math.PI)
  model.translateY(1)
  visitChildren(model, (child) => {
    child.castShadow = true
    child.receiveShadow = true
  })
  scene.add(model))
})
```

And the results look really great:

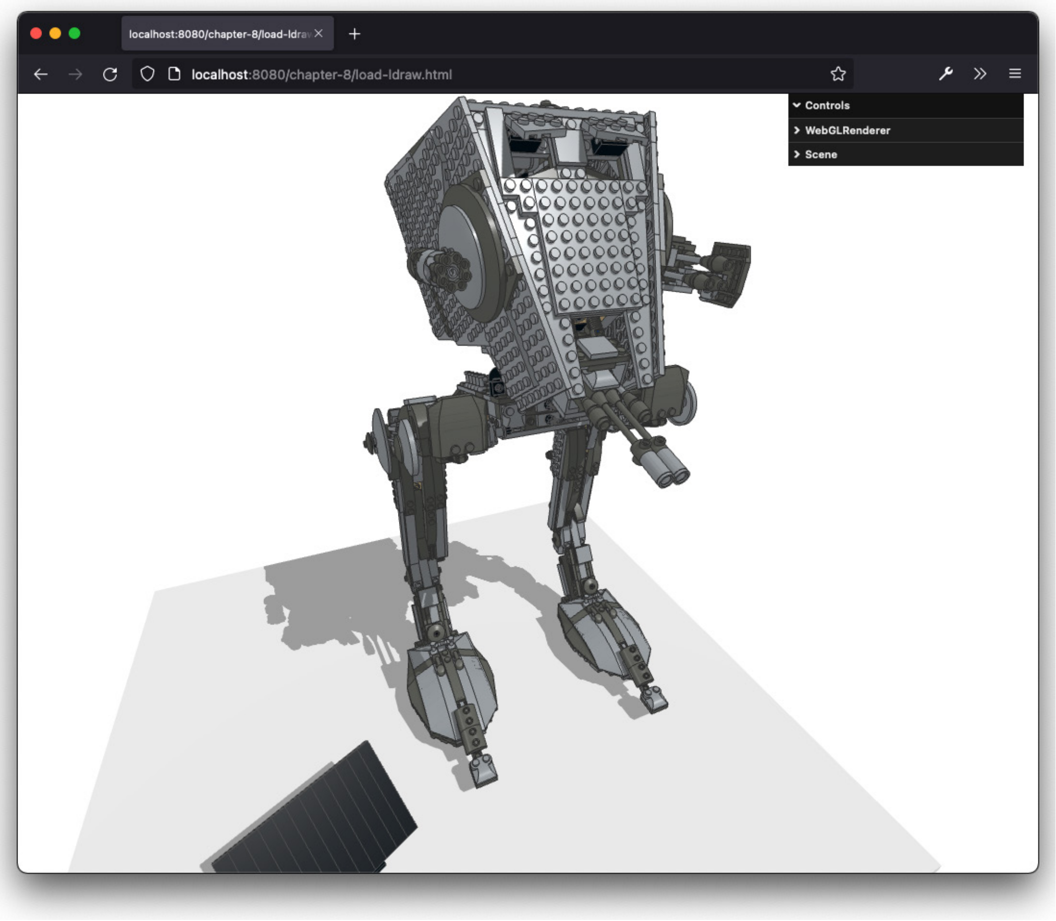

localhost:8080/chapter-8/load-ldra
localhost:8080/chapter-8/load-ldraw.html
Controls
WebGLRenderer
Scene

Figure 8.10 – LEGO Imperial AT-ST model

As you can see, it shows the complete structure of a LEGO set. There are many different models out there that you can use:

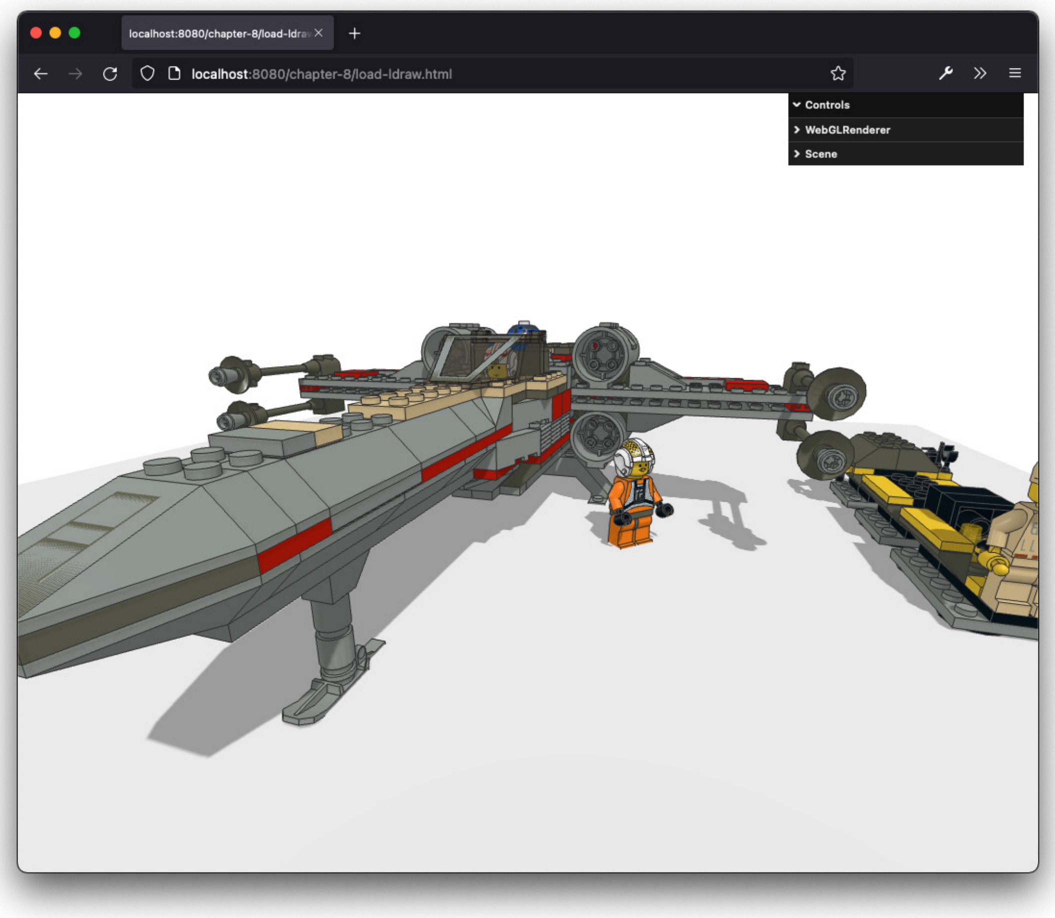

localhost:8080/chapter-8/load-ldra
localhost:8080/chapter-8/load-ldraw.html
Controls
WebGLRenderer
Scene

Figure 8.11 – LEGO X-Wing Fighter

If you want to explore more models, you can download them from the LDraw repository: `https://omr.ldraw.org/`.

Loading voxel-based models

Another interesting approach to creating 3D models is by using voxels. This allows you to build models using small cubes and render them using Three.js. For instance, you can create Minecraft structures outside of Minecraft using such a tool and import them into Minecraft at a later time. A free tool to experiment with voxels is MagicaVoxel (`https://ephtracy.github.io/`). This tool allows you to create voxel models such as this:

Figure 8.12 – Example model created with MagicaVoxel

The interesting part is that you can easily import these models in Three.js using the `VOXLoader` loader, like so:

```
new VOXLoader().loadAsync('/assets/models/vox/monu9.vox').
then((chunks) => {
  const group = new THREE.Group()
  for (let i = 0; i < chunks.length; i++) {
    const chunk = chunks[i]
    const mesh = new VOXMesh(chunk)
    mesh.castShadow = true
    mesh.receiveShadow = true
    group.add(mesh)
  }
  group.scale.setScalar(0.1)
  scene.add(group)
})
```

In the `models` folder, you can find a couple of vox models. The following screenshot shows what it looks like loaded with Three.js:

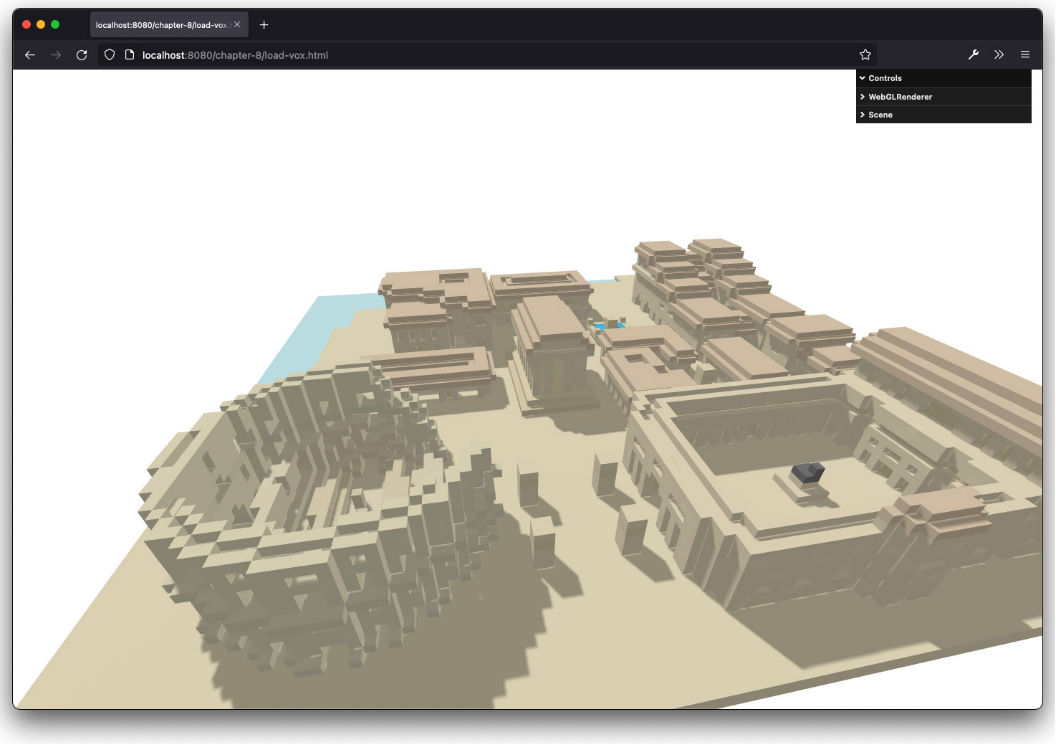

localhost:8080/chapter-8/load-vox.
localhost:8080/chapter-8/load-vox.html
Controls
WebGLRenderer
Scene

Figure 8.13 – Vox model loading with Three.js

The next loader is another very specific one. We'll look at how to render proteins from PDB format.

Showing proteins from PDB

The PDB website (`www.rcsb.org`) contains detailed information about many different molecules and proteins. Besides an explanation of these proteins, it also provides a way to download the structure of these molecules in PDB format. Three.js provides a loader for files specified in the PDB format. In this section, we'll give an example of how you can parse PDB files and visualize them with Three.js.

With this loader included, we're going to create the following 3D model of the molecule description provided (see the `load-pdb.html` example):

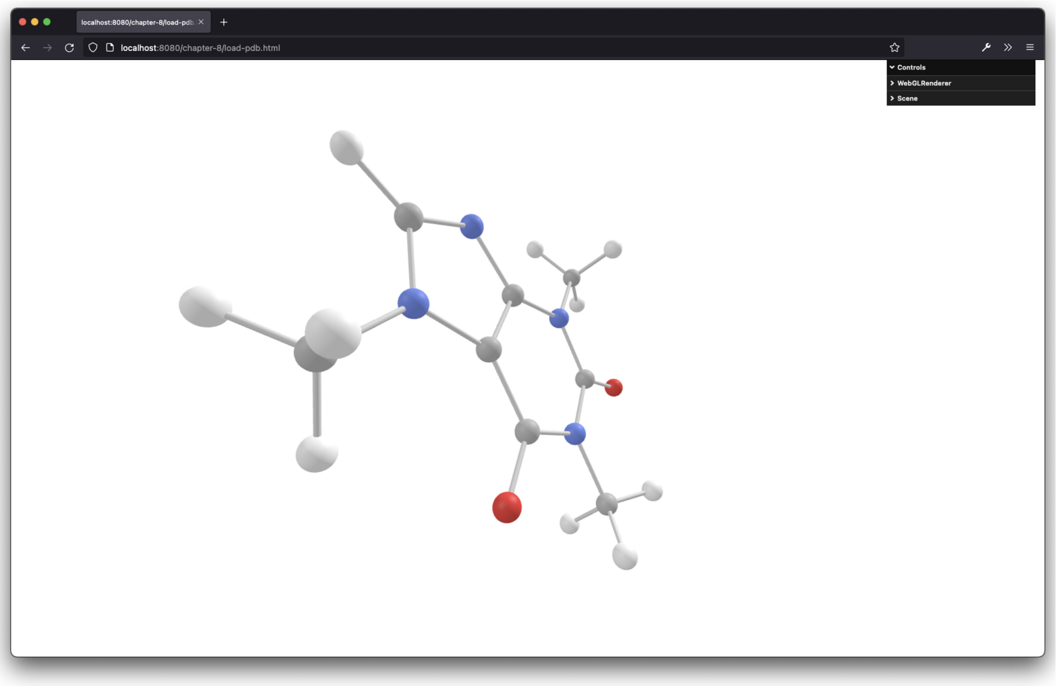

localhost:8080/chapter-8/load-pdb
localhost:8080/chapter-8/load-pdb.html
Controls
WebGLRenderer
Scene

Figure 8.14 – Visualizing a protein using Three.js and PDBLoader

Loading a PDB file is done in the same manner as the previous formats, as follows:

```
PDBLoader().loadAsync('/assets/models/molecules/caffeine.pdb').
then((geometries) => {
  const group = new THREE.Object3D()
  // create the atoms
  const geometryAtoms = geometries.geometryAtoms

  for (let i = 0; i < geometryAtoms.attributes.
    position.count; i++) {
    let startPosition = new THREE.Vector3()
    startPosition.x = geometryAtoms.attributes.
      position.getX(i)
    startPosition.y = geometryAtoms.attributes.
      position.getY(i)
    startPosition.z = geometryAtoms.attributes.position.getZ(i)
    let color = new THREE.Color()
    color.r = geometryAtoms.attributes.color.getX(i)
    color.g = geometryAtoms.attributes.color.getY(i)
    color.b = geometryAtoms.attributes.color.getZ(i)
    let material = new THREE.MeshPhongMaterial({
      color: color
    })
    let sphere = new THREE.SphereGeometry(0.2)
    let mesh = new THREE.Mesh(sphere, material)
    mesh.position.copy(startPosition)
    group.add(mesh)
  }
  // create the bindings
  const geometryBonds = geometries.geometryBonds
  for (let j = 0; j <
    geometryBonds.attributes.position.count; j += 2) {
    let startPosition = new THREE.Vector3()
    startPosition.x = geometryBonds.attributes.
      position.getX(j)
    startPosition.y = geometryBonds.attributes.position.
      getY(j)
    startPosition.z = geometryBonds.attributes.position.
      getZ(j)
    let endPosition = new THREE.Vector3()
    endPosition.x = geometryBonds.attributes.position.
      getX(j + 1)
    endPosition.y = geometryBonds.attributes.position.
      getY(j + 1)
    endPosition.z = geometryBonds.attributes.position.
      getZ(j + 1)
    // use the start and end to create a curve, and use the
      curve to draw
    // a tube, which connects the atoms
    let path = new THREE.CatmullRomCurve3([startPosition,
      endPosition])
    let tube = new THREE.TubeGeometry(path, 1, 0.04)
    let material = new THREE.MeshPhongMaterial({
      color: 0xcccccc
    })
    let mesh = new THREE.Mesh(tube, material)
    group.add(mesh)
  }
  group.scale.set(0.5, 0.5, 0.5)
  scene.add(group)
})
```

As you can see from this example code, we instantiate a `THREE.PDBLoader` object and pass in the model file we want to load, and once the model is loaded, we process it. In this case, the model consists of two properties: `geometryAtoms` and `geometryBonds`. The position attributes from `geometryAtoms` contain the positions of the individual atoms, and the color attributes can be used to color the individual atoms. For a link between the atoms, `geometryBonds` is used.

Based on the position and color, we create a `THREE.Mesh` object and add it to a group:

```
    let sphere = new THREE.SphereGeometry(0.2)
    let mesh = new THREE.Mesh(sphere, material)
    mesh.position.copy(startPosition)
    group.add(mesh)
```

With regard to the connection between the atoms, we follow the same approach. We get the start and end positions of the connection and use those to draw the connection:

```
let path = new THREE.CatmullRomCurve3([startPosition,
  endPosition])
let tube = new THREE.TubeGeometry(path, 1, 0.04)
let material = new THREE.MeshPhongMaterial({
  color: 0xcccccc
})
let mesh = new THREE.Mesh(tube, material)
group.add(mesh)
```

For the connection, we first create a 3D path using `THREE.CatmullRomCurve3`. This path is used as input for `THREE.TubeGeometry` and is used to create a connection between the atoms. All the connections and atoms are added to a group, and this group is added to the scene. There are many models you can download from PDB. For instance, the following screenshot shows the structure of a diamond:

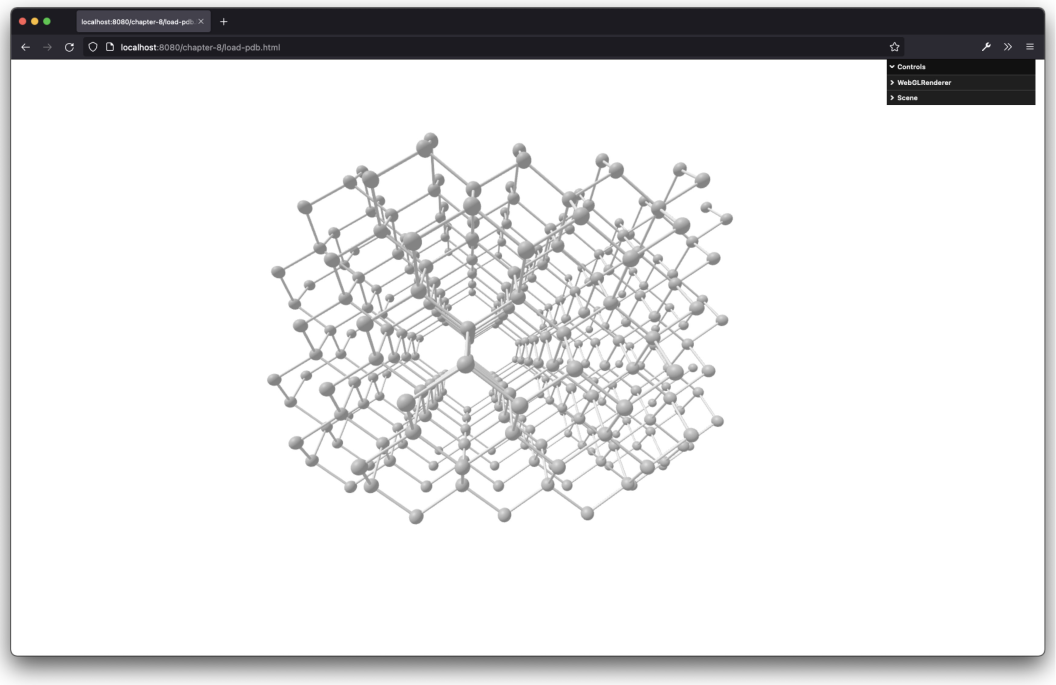

localhost:8080/chapter-8/load-pdb.
localhost:8080/chapter-8/load-pdb.html
Controls
WebGLRenderer
Scene

Figure 8.15 – The structure of a diamond

In the next section, we're looking at the support Three.js has for the PLY model, which can be used to load point cloud data.

Loading a point cloud from a PLY model

Working with the PLY format isn't that much different than the other formats. You include the loader and handle the loaded model. For this last example, however, we're going to do something different. Instead of rendering the model as a mesh, we'll use the information from this model to create a particle system (see the `load-ply.html` example in the following screenshot):

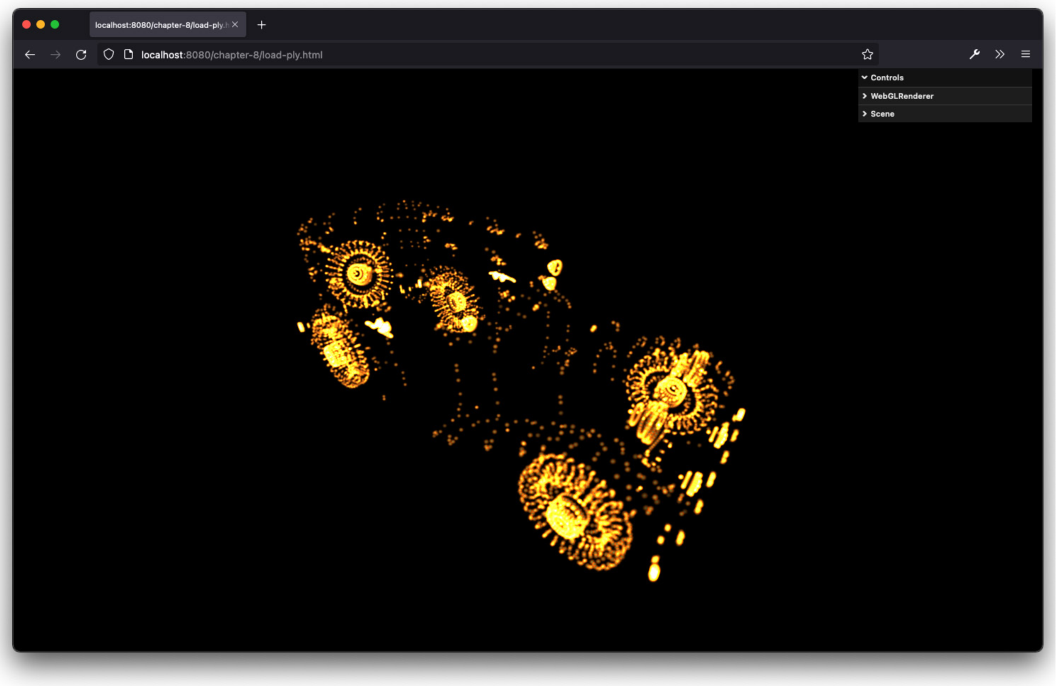

localhost:8080/chapter-8/load-ply.html
Controls
WebGLRenderer
Scene

Figure 8.16 – Point cloud loaded from a PLY model

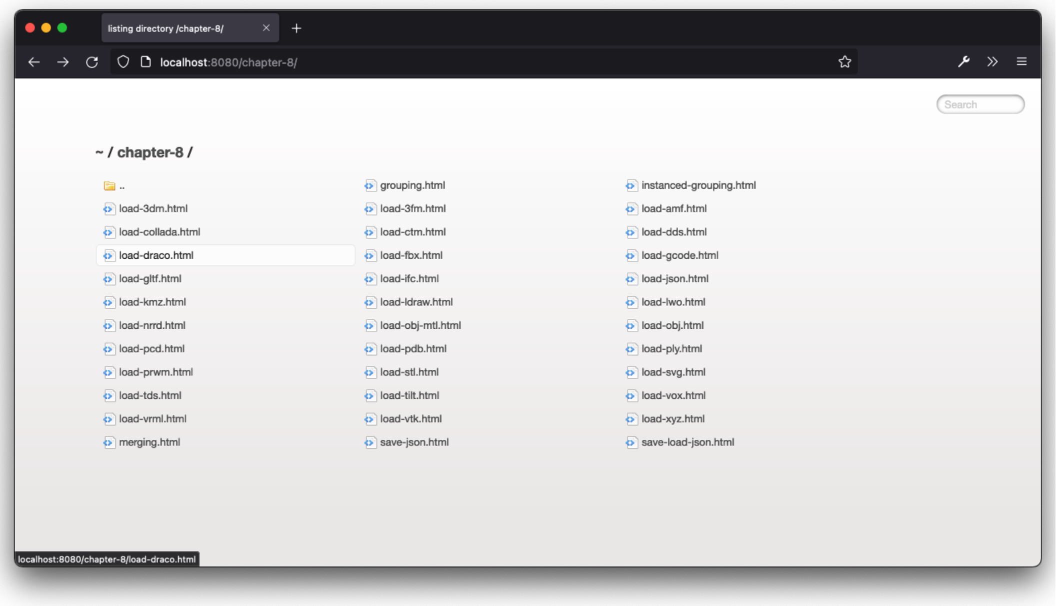
The JavaScript code to render the preceding screenshot is actually very simple; it looks like this:

```
const texture = new THREE.TextureLoader().load('/assets
  /textures/particles/glow.png')
const material = new THREE.PointsMaterial({
  size: 0.15,
  vertexColors: false,
  color: 0xffffff,
  map: texture,
  depthWrite: false,
  opacity: 0.1,
  transparent: true,
  blending: THREE.AdditiveBlending
})
return new PLYLoader().loadAsync('/assets/
  models/carcloud/carcloud.ply').then((model) => {
  const points = new THREE.Points(model, material)
  points.scale.set(0.7, 0.7, 0.7)
  scene.add(points)
})
```

As you can see, we use `THREE.PLYLoader` to load the model and use this geometry as input for `THREE.Points`. The material we use is the same as what we used for the last example in *Chapter 7*, *Points and Sprites*. As you can see, with Three.js, it is very easy to combine models from various sources and render them in different ways, all with a few lines of code.

Other loaders

At the beginning of this chapter, in the *Loading geometries from external resources* section, we showed you a list of all the different loaders provided by Three.js. We've provided examples of all these in the sources for `chapter-8`:


listing directory /chapter-8/
localhost:8080/chapter-8/
Search
localhost:8080/chapter-8/load-draco.html

~ / chapter-8 /

..
load-3dm.html
load-collada.html
load-draco.html
load-gltf.html
load-kmz.html
load-nrrd.html
load-pcd.html
load-prwm.html
load-tds.html
load-vrml.html
merging.html
grouping.html
load-3fm.html
load-ctm.html
load-fbx.html
load-ifc.html
load-ldraw.html
load-obj-mtl.html
load-pdb.html
load-stl.html
load-tilt.html
load-vtk.html
save-json.html
instanced-grouping.html
load-amf.html
load-dds.html
load-gcode.html
load-json.html
load-lwo.html
load-obj.html
load-ply.html
load-svg.html
load-vox.html
load-xyz.html
save-load-json.html

Figure 8.17 – Directory showing examples of all the loaders

The source code for all these loaders follows the same patterns as we've seen for the loaders we have explained in this chapter. Just load the model, determine which part of the loaded model you want to show, make sure scaling and positions are correct, and add it to the scene.

Summary

Using models from external sources isn't that hard to do in Three.js, especially for simple models—you only have to take a few easy steps.

When working with external models, or creating them using grouping and merging, it is good to keep a couple of things in mind. The first thing you need to remember is that when you group objects, they remain available as individual objects. Transformations applied to the parent also affect the children, but you can still transform the children individually. Besides grouping, you can also merge geometries together. With this approach, you lose the individual geometries and get a single new geometry. This is especially useful when you're dealing with thousands of geometries you need to render and you're running into performance issues. The final approach if you want to control a large number of meshes of the same geometry is to use a `THREE.InstancedMesh` object or a `THREE.InstancedBufferGeometry` object, which allows you to position and transform the individual meshes, but still get great performance.

Three.js supports a large number of external formats. When using these format loaders, it's a good idea to look through the source code and add `console.log` statements to determine what the data loaded really looks like. This will help you to understand the steps you need to take to get the correct mesh and set it to the correct position and scale. Often, when the model doesn't show correctly, this is caused by its material settings. It could be that incompatible texture formats are used, opacity is incorrectly defined, or the format contains incorrect links to the texture images. It is usually a good idea to use a test material to determine whether the model itself is loaded correctly and log the loaded material to the JavaScript console to check for unexpected values.

If you want to reuse your own scenes or models, you can simply export these by just calling the `asJson` function and loading them again with `ObjectLoader`.

The models you worked with in this chapter, and in the previous chapters, are mostly static models. They aren't animated, don't move around, and don't change shape. In *Chapter 9*, you'll learn how you can animate your models to make them come to life. Besides animations, the next chapter will also explain the various camera controls provided by Three.js. With a camera control, you can move, pan, and rotate the camera around your scene.

9

Animation and Moving the Camera

In the previous chapters, we saw some simple animations, but nothing too complex. In *Chapter 1*, *Creating Your First 3D Scene with Three.js*, we introduced the basic rendering loop, and in the chapters following that, we used that to rotate some simple objects and show a couple of other basic animation concepts.

In this chapter, we're going to look in more detail at how animation is supported by Three.js. We will look at the following four subjects:

- Basic animations
- Working with the camera
- Morphing and skeleton animation
- Creating animations using external modes

We will start by covering the basic concepts behind animations.

Basic animations

Before we look at the examples, let's do a quick recap of what was shown in *Chapter 1*, on the render loop. To support animations, we need to tell Three.js to render the scene every so often. For this, we use the standard HTML5 `requestAnimationFrame` functionality, as follows:

```
function animate() {
  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();
```

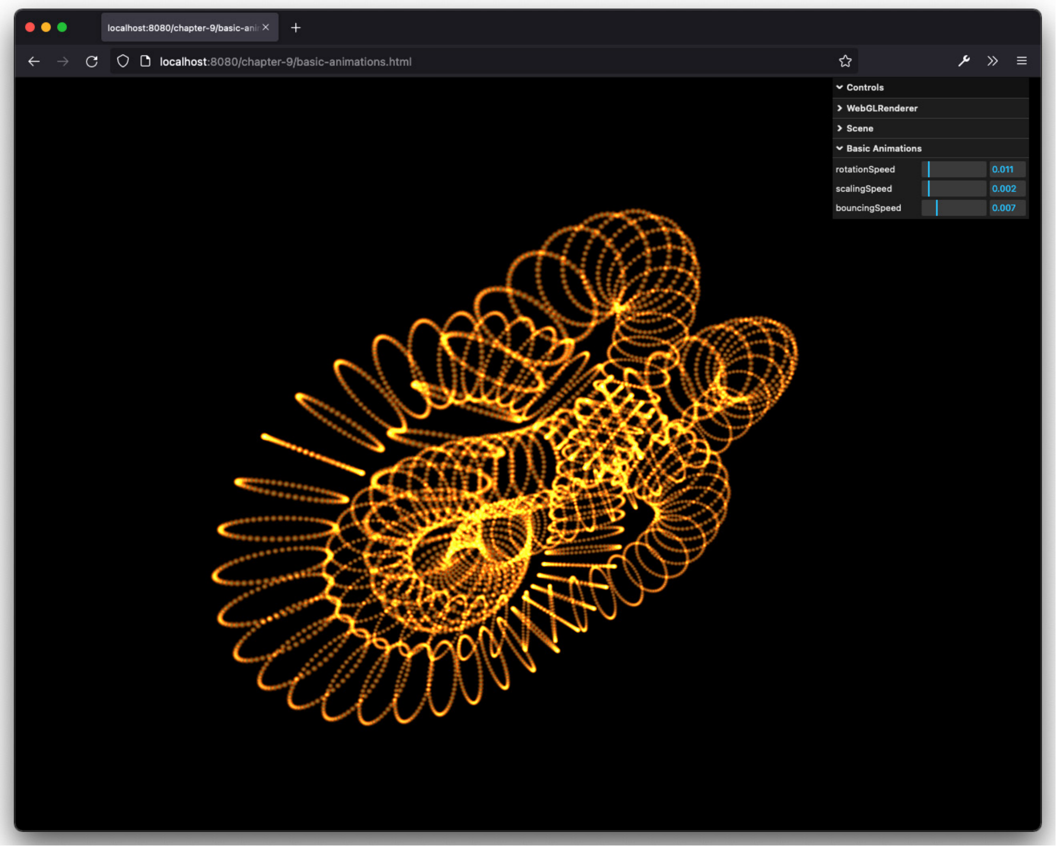
With this code, we only need to call the `render()` function once we've initialized the scene. In the `render()` function itself, we use `requestAnimationFrame` to schedule the next rendering. This way, the browser will make sure the `render()` function is called at the correct interval (usually around 60 times or 120 times a second). Before `requestAnimationFrame` was added to browsers, `setInterval(function, interval)` or `setTimeout(function, interval)` was used. These would call the specified function once every set interval.

The problem with this approach is that it doesn't take into account what else is going on. Even if your animation isn't shown or is in a hidden tab, it is still called and is still using resources. Another issue is that these functions update the screen whenever they are called, and not when it is the best time for the browser, which results in higher CPU usage. With `requestAnimationFrame`, we don't tell the browser when it needs to update the screen; we ask the browser to run the supplied function when it's most opportune. Usually, this results in a frame rate of about 60 or 120 FPS (depending on your hardware). With `requestAnimationFrame`, your animations will run more smoothly and will be more CPU- and GPU-friendly, and you don't have to worry about timing issues.

In the next section, we'll start with creating a simple animation.

Simple animations

With this approach, we can very easily animate objects by changing their `rotation`, `scale`, `position`, `material`, vertices, faces, and anything else you can imagine. In the next render loop, Three.js will render the changed properties. A very simple example, based on the one we already saw in *Chapter 7*, *Points and Sprites*, is available in `01-basic-animations.html`. The following screenshot shows this example:


localhost:8080/chapter-9/basic-ani
localhost:8080/chapter-9/basic-animations.html
Controls
WebGLRenderer
Scene
Basic Animations

rotationSpeed 0.011
scalingSpeed 0.002
bouncingSpeed 0.007

Figure 9.1 – Animation after changing its properties

The render loop for this is very simple. First, we initialize the various properties on the `userData` object, which is a place for custom data stored in the `THREE.Mesh` itself, and then update these properties on the mesh, using the data we defined on the `userData` object. In the animation loop, just change the rotation, position, and the scale based on these properties, and Three.js handles the rest. Here's how we do this:

```
const geometry = new THREE.TorusKnotGeometry(2, 0.5, 150, 50,
3, 4)
const material = new THREE.PointsMaterial({
  size: 0.1,
  vertexColors: false,
  color: 0xffffff,
  map: texture,
  depthWrite: false,
  opacity: 0.1,
  transparent: true,
  blending: THREE.AdditiveBlending
})
const points = new THREE.Points(geometry, material)
points.userData.rotationSpeed = 0
points.userData.scalingSpeed = 0
points.userData.bouncingSpeed = 0
points.userData.currentStep = 0
points.userData.scalingStep = 0
// in the render loop
function render() {
  const rotationSpeed = points.userData.rotationSpeed
  const scalingSpeed = points.userData.scalingSpeed
  const bouncingSpeed = points.userData.bouncingSpeed
  const currentStep = points.userData.currentStep
  const scalingStep = points.userData.scalingStep
  points.rotation.x += rotationSpeed
  points.rotation.y += rotationSpeed
  points.rotation.z += rotationSpeed
  points.userData.currentStep = currentStep + bouncingSpeed
  points.position.x = Math.cos(points.userData.currentStep)
  points.position.y = Math.abs(Math.sin
   (points.userData.currentStep)) * 2
  points.userData.scalingStep = scalingStep + scalingSpeed
  var scaleX = Math.abs(Math.sin(scalingStep * 3 + 0.5 *
    Math.PI))
  var scaleY = Math.abs(Math.cos(scalingStep * 2))
  var scaleZ = Math.abs(Math.sin(scalingStep * 4 + 0.5 *
    Math.PI))
  points.scale.set(scaleX, scaleY, scaleZ)
}
```

There's nothing spectacular here, but it nicely shows the concept behind the basic animations we will discuss in this book. We just change the `scale`, `rotation`, and `position` properties and Three.js does the rest.

In the next section, we'll take a quick sidestep. Besides animations, an important aspect that you'll quickly run into when working with Three.js in more complex scenes is the ability to select objects on the screen using the mouse.

Selecting and moving objects

Even though not directly related to animations, since we'll be looking at cameras and animations in this chapter, knowing how to select and move objects is a nice addition to the subjects explained in this chapter. Here, we will show you how to do the following:

- Select an object from a scene using the mouse
- Drag an object around the scene with the mouse

We'll start by looking at the steps you need to take to select an object.

Selecting objects

First, open the `selecting-objects.html` example, where you'll see the following:

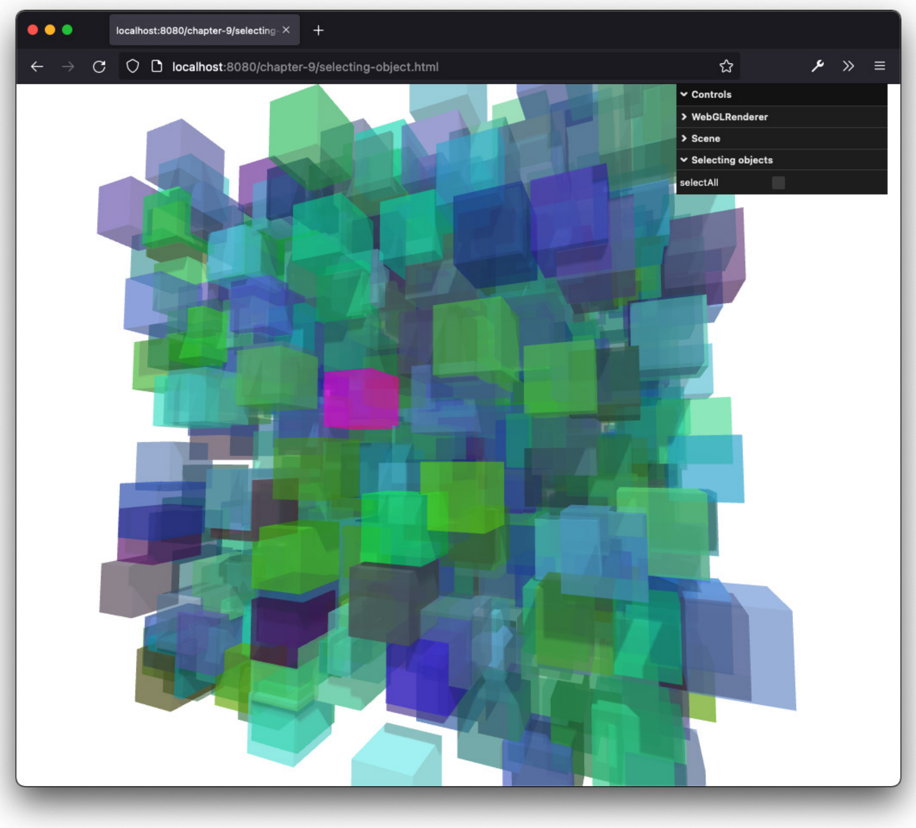

localhost:8080/chapter-9/selecting
localhost:8080/chapter-9/selecting-object.html
Controls
WebGLRenderer
Scene
Selecting objects
selectAll

Figure 9.2 – Randomly placed cubes that can be selected with the mouse

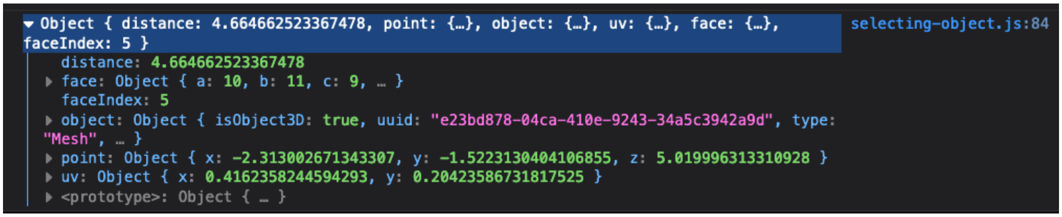
When you move the mouse around the scene, you'll see that whenever your mouse hits an object, that object is highlighted. You can easily create this by using a `THREE.Raycaster`. A raycaster will look at your current camera and cast a ray from the camera to your mouse's position. Based on that, it can calculate which object is hit based on the position of the mouse. To accomplish this, we need to take the following steps:

- Create an object that keeps track of where the mouse is pointing at
- Whenever we move the mouse, update that object
- In the render loop, use this updated information to see which Three.js object we're pointing at

This is shown in the following code fragment:

```
// initially set the position to -1, -1
let pointer = {
  x: -1,
  y: -1
}
// when the mouse moves update the point
document.addEventListener('mousemove', (event) => {
  pointer.x = (event.clientX / window.innerWidth) * 2 - 1
  pointer.y = -(event.clientY / window.innerHeight) * 2 + 1
})
// an array containing all the cubes in the scene
const cubes = ...
// use in the render loop to determine the object to highlight
const raycaster = new THREE.Raycaster()
function render() {
  raycaster.setFromCamera(pointer, camera)
  const cubes = scene.getObjectByName('group').children
  const intersects = raycaster.intersectObjects(cubes)
  // do something with the intersected objects
}
```

Here, we are using `THREE.Raycaster` to determine which objects intersect the position of the mouse, from the point of the camera. The result (`intersects`, in the preceding example) contains all the cubes that intersected our mouse because the ray is cast from the camera's position through to the end of the camera's range. The first one in this array is the one that we're hovering over, and the other values in this array (if any) point to objects behind the first mesh. `THREE.Raycaster` also provides other information about exactly where you hit the object:



```
▾ Object { distance: 4.664662523367478, point: {…}, object: {…}, uv: {…}, face: {…},      selecting-object.js:84
faceIndex: 5 }
    distance: 4.664662523367478
  ▸ face: Object { a: 10, b: 11, c: 9, … }
    faceIndex: 5
  ▸ object: Object { isObject3D: true, uuid: "e23bd878-04ca-410e-9243-34a5c3942a9d", type:
  "Mesh", … }
  ▸ point: Object { x: -2.313002671343307, y: -1.5223130404106855, z: 5.019996313310928 }
  ▸ uv: Object { x: 0.4162358244594293, y: 0.20423586731817525 }
  ▸ <prototype>: Object { … }
```

Figure 9.3 – Additional information from the raycaster

Here, we clicked on the `face` object. `faceIndex` points to the face of the mesh that was selected. The `distance` value is measured from the camera to the clicked object, and `point` is the exact position on the mesh where it was clicked. Finally, we have the `uv` value, which determines, when using textures, where the point that was clicked appears on the 2D texture (ranging from 0 to 1; more information on `uv` can be found in *Chapter 10*, *Loading and Working With Textures*).

Dragging objects

Besides selecting an object, a common requirement is being able to drag and move objects around. Three.js also provides default support for this. If you open the `dragging-objects.html` example in your browser, you'll see a similar scene to the one shown in *Figure 9.2*. This time, when you click on an object, you can drag it around the scene:

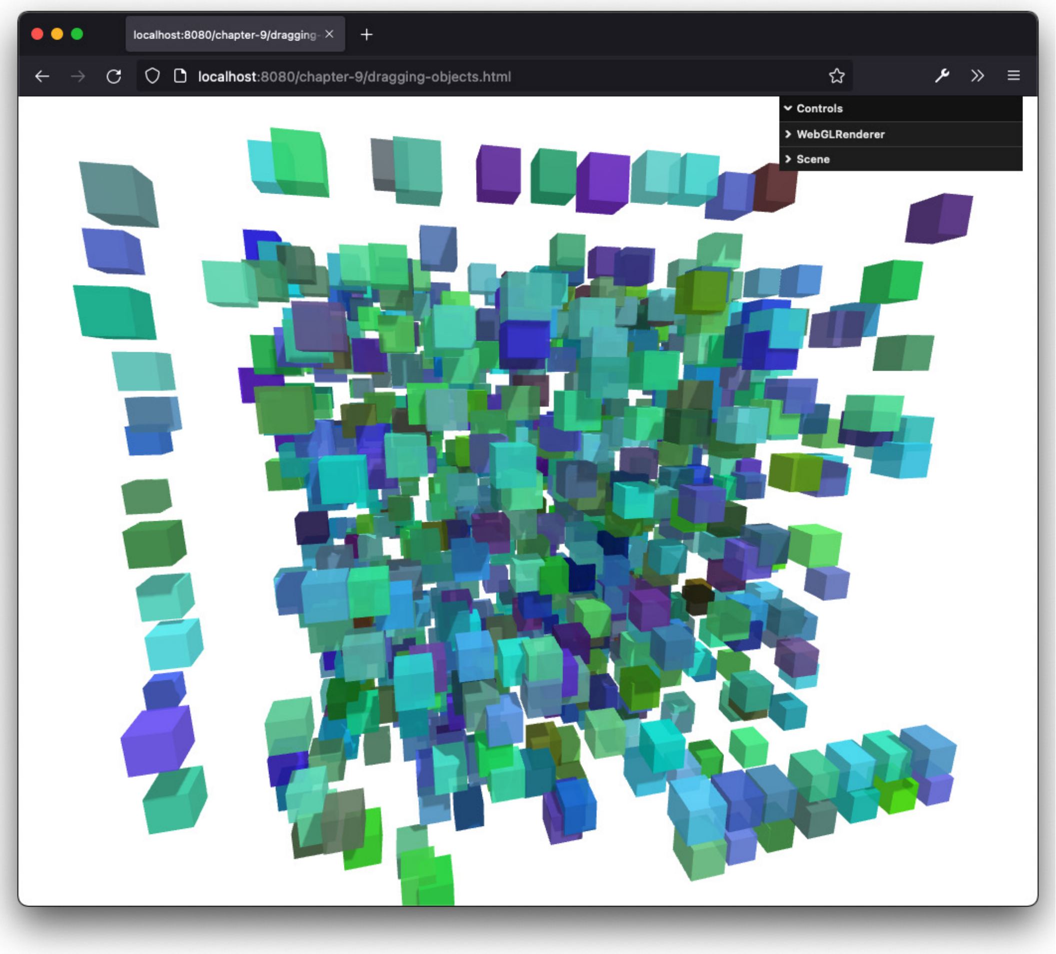

localhost:8080/chapter-9/dragging-
localhost:8080/chapter-9/dragging-objects.html
Controls
WebGLRenderer
Scene

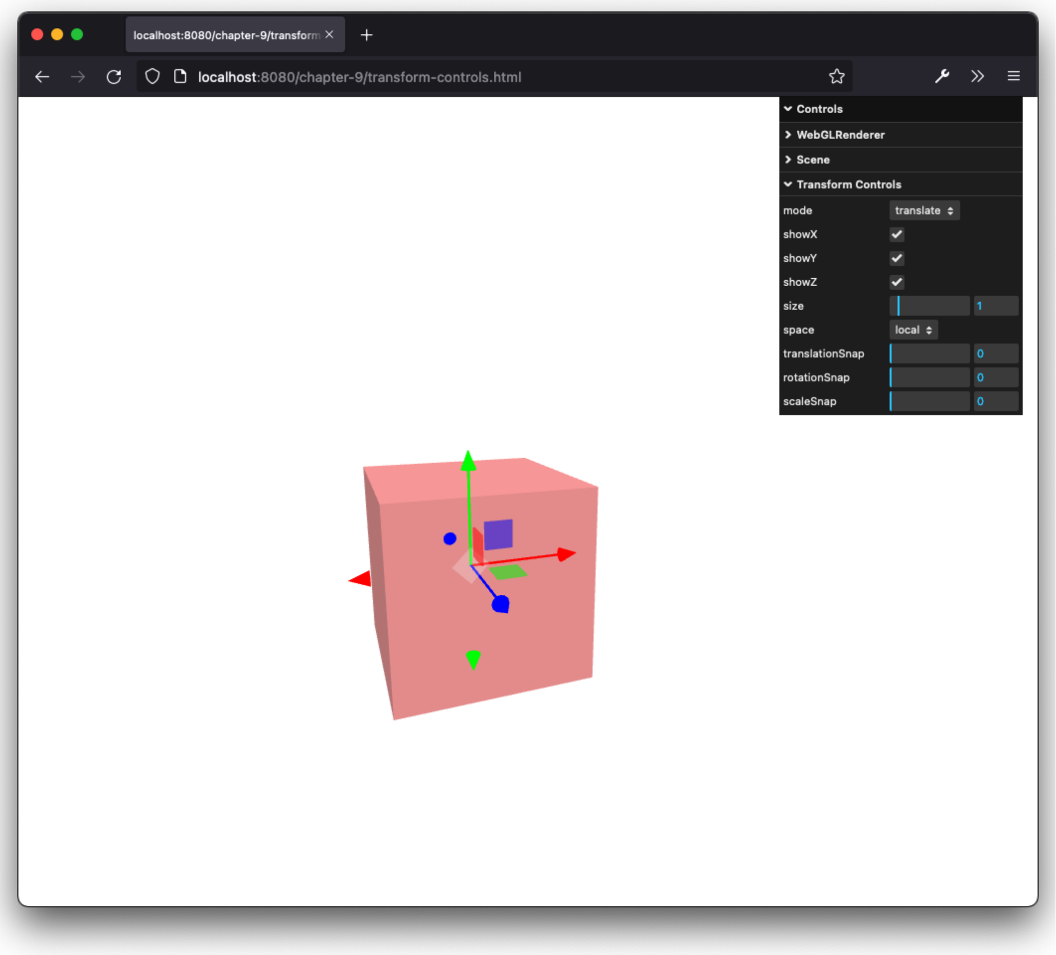
Figure 9.4 – Dragging an object around the scene using the mouse

To support dragging objects, Three.js uses something called `DragControls`. This handles everything and provides convenient callbacks whenever the dragging starts and stops. The code to accomplish this is shown here:

```
const orbit = new OrbitControls(camera, renderer.domElement)
orbit.update()
const controls = new DragControls(cubes, camera, renderer.
domElement)
controls.addEventListener('dragstart', function (event) {
  orbit.enabled = false
  event.object.material.emissive.set(0x33333)
})
controls.addEventListener('dragend', function (event) {
  orbit.enabled = true
  event.object.material.emissive.set(0x000000)
})
```

It is as simple as that. Here, we added `DragControls` and passed in the elements that can be dragged (in our case, all of the randomly placed cubes). Then, we added two event listeners. The first one, `dragstart`, is called when we start dragging a cube, whereas `dragend` is called when we stop dragging an object. In this example, when we start dragging, we disable `OrbitControls` (which allows us to use the mouse to look around the scene) and change the color of the selected object. Once we stop dragging, we change the color of the object back and enable `OrbitControls` again.

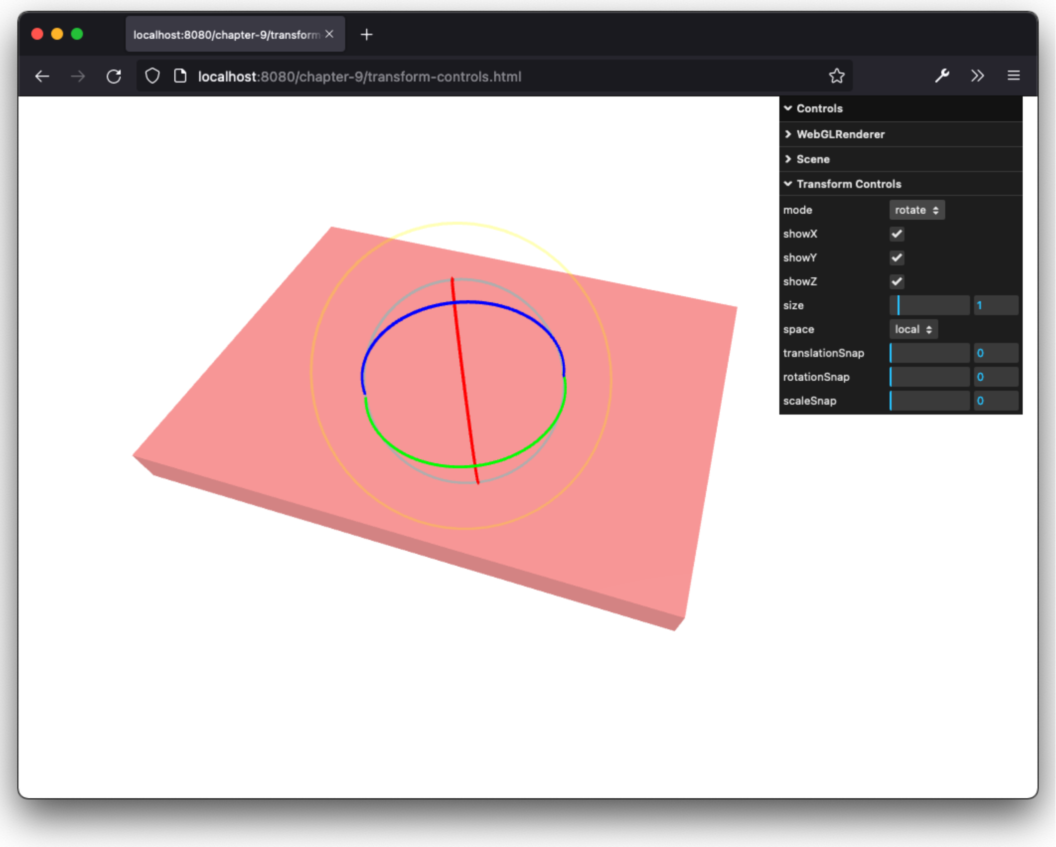
There is also a somewhat more advanced version of `DragControls` available called `TransformControls`. We won't go into the details of this control, but it allows you to use a simple UI to transform the properties of a mesh. You can find an example of this control when you open `transform-controls-html` in your browser:


localhost:8080/chapter-9/transform
localhost:8080/chapter-9/transform-controls.html
Controls
WebGLRenderer
Scene
Transform Controls

mode: translate
showX: ✓
showY: ✓
showZ: ✓
size: 1
space: local
translationSnap: 0
rotationSnap: 0
scaleSnap: 0

Figure 9.5 – Transform controls allow you to change the properties of a mesh

If you click on the various parts of this control, you can easily change the shape of the cube:


localhost:8080/chapter-9/transform
localhost:8080/chapter-9/transform-controls.html
Controls
WebGLRenderer
Scene
Transform Controls

mode rotate
showX ✓
showY ✓
showZ ✓
size 1
space local
translationSnap 0
rotationSnap 0
scaleSnap 0

Figure 9.6 – Shape modified using transform controls

For the final example in this chapter, we'll show you how you can use an alternative way of modifying the properties of an object (as we saw in the first example of this chapter) by using a tweening library.

Animating with Tween.js

Tween.js is a small JavaScript library that you can download from `https://github.com/sole/tween.js/` and that you can use to easily define the transition of a property between two values. All the intermediate points between the start and end values are calculated for you. This process is called **tweening**. For instance, you can use this library to change the `x` position of a mesh from 10 to 3 in 10 seconds, as follows:

```
const tween = new TWEEN.Tween({x: 10}).to({x: 3}, 10000)
.easing(TWEEN.Easing.Elastic.InOut)
.onUpdate( function () {
  // update the mesh
})
```

Alternatively, you can create a separate object and pass that into the mesh you want to work with:

```
const tweenData = {
  x: 10
}
new TWEEN.Tween(tweenData)
  .to({ x: 3 }, 10000)
  .yoyo(true)
  .repeat(Infinity)
  .easing(TWEEN.Easing.Bounce.InOut)
  .start()
mesh.userData.tweenData = tweenData
```

In this example, we've created `TWEEN.Tween`. This tween will make sure that the `x` property is changed from 10 to 3 over 10,000 milliseconds. Tween.js also allows you to define how this property is changed over time. This can be done using linear, quadratic, or any of the other possibilities (see `http://sole.github.io/tween.js/examples/03_graphs.html` for a complete overview). The value is changed over time by a process called **easing**. With Tween.js, you configure this using the `easing()` function. This library also provides additional ways to control how this easing is done. For instance, we can set how often the easing should be repeated (`repeat(10)`) and whether we want a yoyo effect (this means we go from 10 to 3 and back to 10 in this example).

Using this library together with Three.js is very simple. If you open the `tween-animations.html` example, you will see the Tween.js library in action. The following screenshot shows a still image of the example:

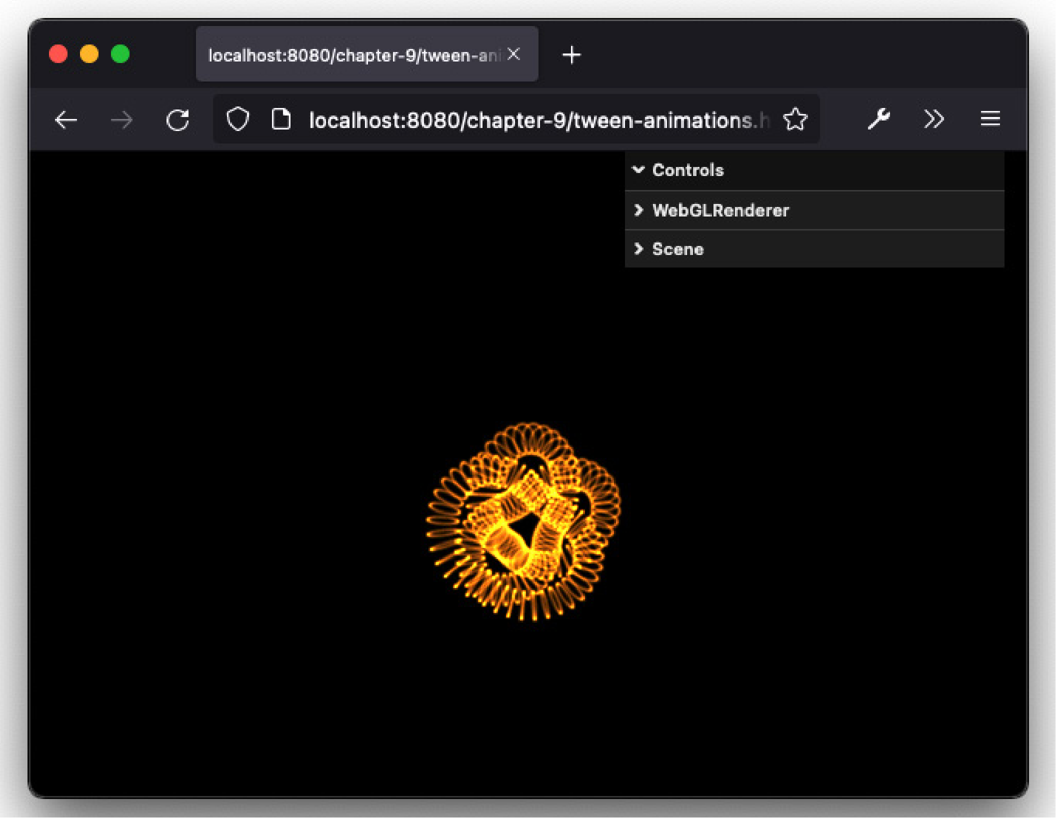

localhost:8080/chapter-9/tween-ani
localhost:8080/chapter-9/tween-animations.h
Controls
WebGLRenderer
Scene

Figure 9.7 – Tweening a point system halfway through the action

We'll use the Tween.js library to move this to a single point using a specific `easing()`, which at a certain point looks as follows:

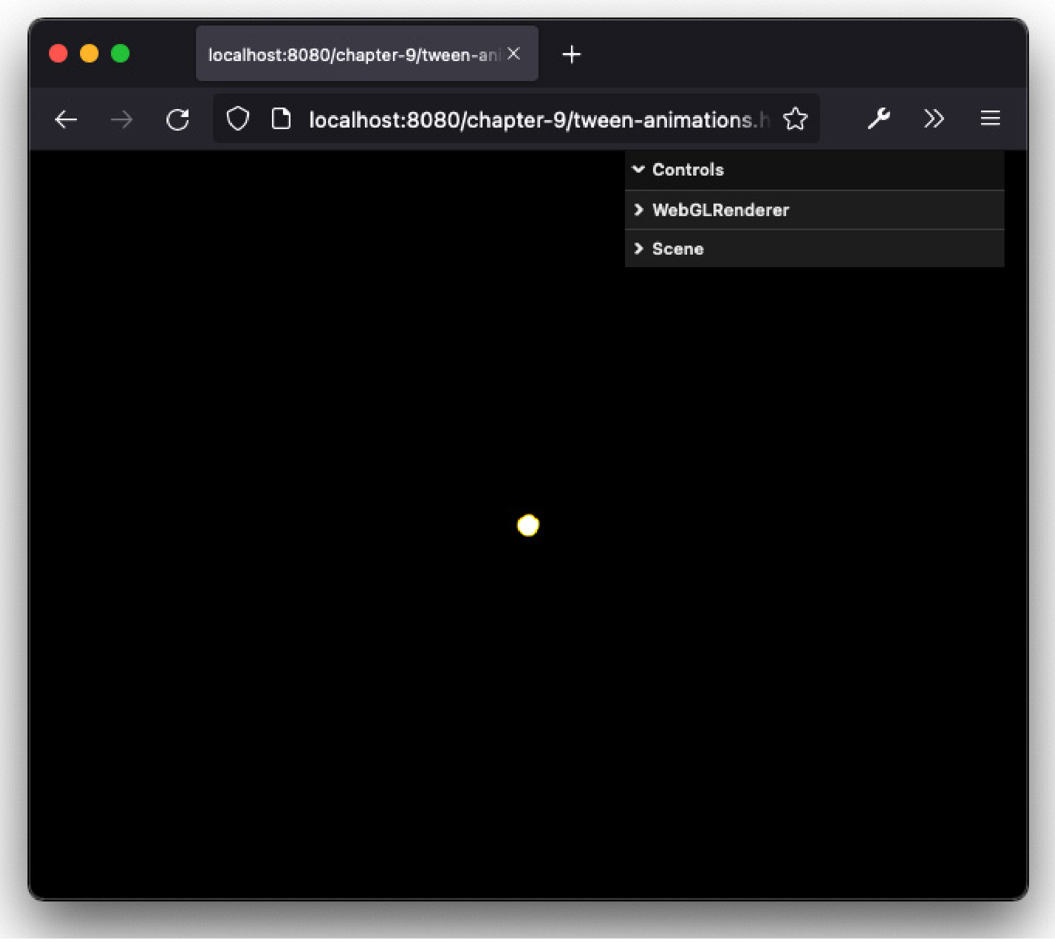

localhost:8080/chapter-9/tween-ani
localhost:8080/chapter-9/tween-animations.h
Controls
WebGLRenderer
Scene

Figure 9.8 – Tweening a point when everything is merged into a single point

In this example, we've taken a point cloud from *Chapter 7*, and created an animation where all the points slowly move down to the center. The position of these particles is set by using a tween created with the Tween.js library, as follows:

```
const geometry = new THREE.TorusKnotGeometry(2, 0.5, 150, 50,
3, 4)
geometry.setAttribute('originalPos', geometry.
attributes['position'].clone())
const material = new THREE.PointsMaterial(..)
const points = new THREE.Points(geometry, material)
const tweenData = {
  pos: 1
}
new TWEEN.Tween(tweenData)
  .to({ pos: 3 }, 10000)
  .yoyo(true)
  .repeat(Infinity)
  .easing(TWEEN.Easing.Bounce.InOut)
  .start()
points.userData.tweenData = tweenData
// in the render loop
const originalPosArray = points.geometry.attributes.
originalPos.array
const positionArray = points.geometry.attributes.position.array
TWEEN.update()
 for (let i = 0; i < points.geometry.attributes.position.count;
i++) {
  positionArray[i * 3] = originalPosArray[i * 3] * points.
userData.tweenData.pos
  positionArray[i * 3 + 1] = originalPosArray[i * 3 + 1] *
points.userData.tweenData.pos
  positionArray[i * 3 + 2] = originalPosArray[i * 3 + 2] *
points.userData.tweenData.pos
}
points.geometry.attributes.position.needsUpdate = true
```

With this piece of code, we created a tween that transitions a value from `1` to `0` and back again. To use the value from the tween, we have two different options: we can use the `onUpdate` function provided by this library to call a function with the updated values, whenever the tween is updated (which is done by calling `TWEEN.update()`), or we can directly access the updated values. In this example, we used the latter approach.

Before we look at the changes we need to make in the `render` function, we must perform one additional step after we load the model. We want to tween between the original values to zero and back again. For this, we need to store the original positions of the vertices somewhere. We can do this by copying the starting positions array:

```
geometry.setAttribute('originalPos', geometry.
attributes['position'].clone())
```

Now, whenever we want to access the original position, we can look at the `originalPos` attribute on the geometry. Now, we can just use the value from the tween to calculate the new positions of each of the vertices. We can do this like so in the render loop:

```
const originalPosArray = points.geometry.attributes.
originalPos.array
const positionArray = points.geometry.attributes.position.array
for (let i = 0; i < points.geometry.attributes.position.count;
i++) {
  positionArray[i * 3] = originalPosArray[i * 3] * points.
userData.tweenData.pos
  positionArray[i * 3 + 1] = originalPosArray[i * 3 + 1] *
points.userData.tweenData.pos
  positionArray[i * 3 + 2] = originalPosArray[i * 3 + 2] *
points.userData.tweenData.pos
}
points.geometry.attributes.position.needsUpdate = true
```

With these steps in place, the tween library will take care of positioning the various points on the screen. As you can see, using this library is much easier than having to manage the transitions yourself. Besides animating and changing objects, we can also animate a scene by moving the camera around. In the previous chapters, we did this a couple of times by manually updating the position of the camera. Three.js also provides several additional ways of updating the camera.

Working with the camera

Three.js has several camera controls you can use to control the camera throughout a scene. These controls are located in the Three.js distribution and can be found in the `examples/js/controls` directory. In this section, we'll look at the following controls in more detail:

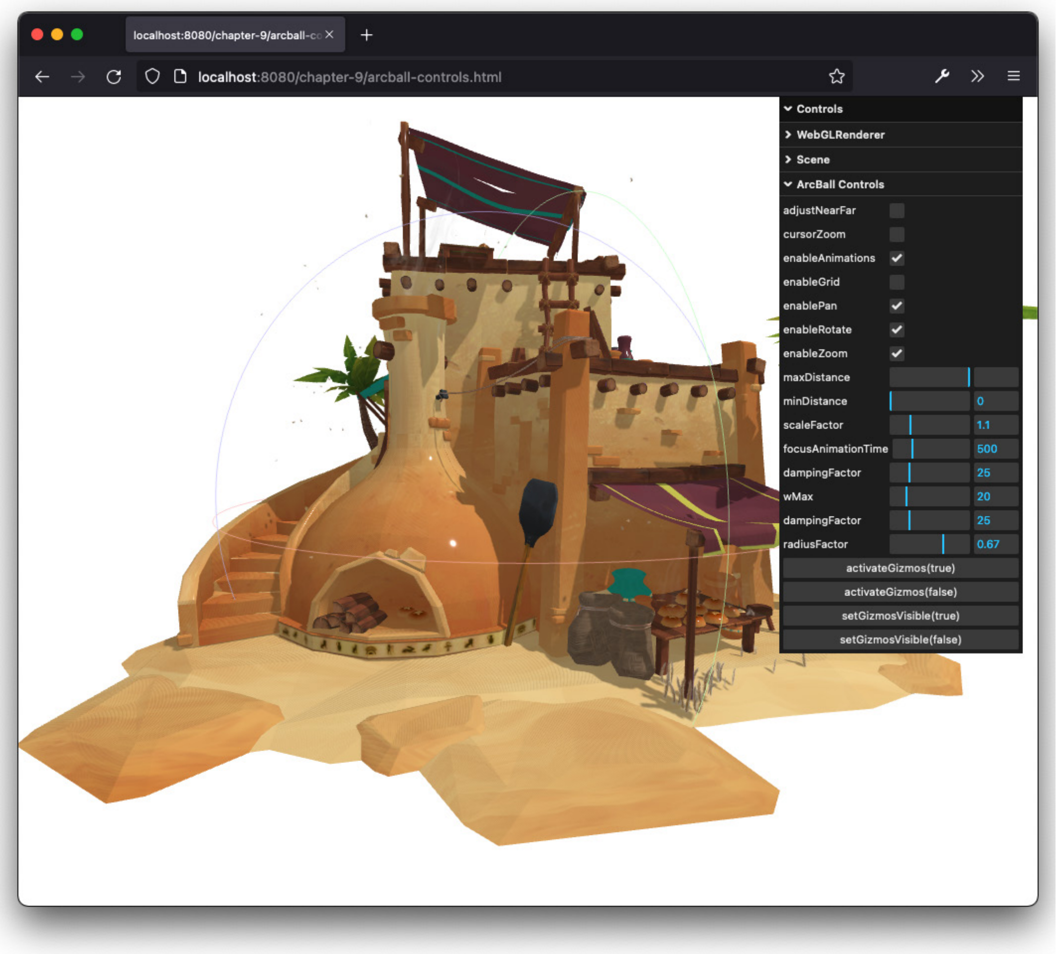
- `ArcballControls`: An extensive control that provides a transparent overlay that you can use to easily move the camera around.
- `FirstPersonControls`: These are controls that behave like those in first-person shooters. You can move around with the keyboard and look around with the mouse.
- `FlyControls`: These are flight simulator-like controls. You can move and steer with the keyboard and the mouse.
- `OrbitControls`: This simulates a satellite in orbit around a specific scene. This allows you to move around with the mouse and keyboard.
- `PointerLockControls`: These are similar to the first-person controls but they also lock the mouse pointer to the screen, making it a great choice for simple games.
- `TrackBallControls`: These are the most-used controls, allowing you to use the mouse (or the trackball) to easily move, pan, and zoom around the scene.

Besides using these camera controls, you can also move the camera yourself by setting its position and changing where it is pointed using the `lookAt()` function.

The first control we'll look at is `ArcballControls`.

ArcballControls

The easiest way to explain how `ArcballControls` works is by looking at an example. If you open up the `arcball-controls.html` example, you'll see a simple scene, like this:



localhost:8080/chapter-9/arcball-controls.html

- Controls
  - WebGLRenderer
  - Scene
  - ArcBall Controls
    - adjustNearFar ☐
    - cursorZoom ☐
    - enableAnimations ☑
    - enableGrid ☐
    - enablePan ☑
    - enableRotate ☑
    - enableZoom ☑
    - maxDistance
    - minDistance 0
    - scaleFactor 1.1
    - focusAnimationTime 500
    - dampingFactor 25
    - wMax 20
    - dampingFactor 25
    - radiusFactor 0.67

Figure 9.9 – Using ArcballControls to explore a scene

If you look closely at this screenshot, you will see two translucent lines crossing the scene. These are lines provided by `ArcballControls`, which you can use to rotate and pan around the scene. These lines are called **gizmos**. The left mouse button is used to rotate the scene, the right mouse button can be used to pan around, and you can zoom in with the scroll wheel.

Besides this standard functionally, this control also allows you to focus on specific parts of the mesh that is shown. If you double-click on the scene, the camera will focus on that part of the scene. To use this control, all we need to do is instantiate it and pass in the `camera` property, the `domElement` property used by the renderer, and the `scene` property we're looking at:

```
import { ArcballControls } from 'three/examples/jsm/controls/
ArcballControls'
```

```
const controls = new ArcballControls(camera, renderer.
domElement, scene)
controls.update()
```

This control is a very versatile one, which can be configured through a set of properties. Most of these properties can be explored in this example by using the menu on the right of this example. For this specific control, we'll dive a bit deeper into the properties and methods provided by this object since it is a versatile control and a good choice when you want to provide a good way for your users to explore your scenes. Let's provide an overview of the properties and the methods provided by this control. First, let's look at the properties:

- `adjustNearFar`: If this is set to `true`, this control will change the camera's `near` and `far` properties when zooming in
- `camera`: The camera that's used when creating this control
- `cursorZoom`: If set to `true`, when zooming in, the zoom will be focused on the position of the cursor
- `dampingFactor`: If `enableAnimations` is set to `true`, this value will determine how quickly an animation stops after an action
- `domElement`: This element is used to list mouse events
- `enabled`: Determines whether this control is enabled or not
- `enableRotate`, `enableZoom`, `enablePan`, `enableGrid`, `enableAnimations`: These properties enable and disable functionality provided by this control
- `focusAnimationTime`: When we double-click and focus on part of the scene, this property determines the duration of the focusing animation
- `maxDistance`/`minDistance`: How far we can zoom out and in for `PerspectiveCamera`
- `maxZoom`/`minZoom`: How far we can zoom out and in for `OrthographicCamera`
- `scaleFactor`: How fast we zoom in and out
- `scene`: The scene passed in the constructor
- `radiusFactor`: The size of the "gizmo" relative to the screen's width and height
- `wMax`: How fast we're allowed to rotate the scene

This control also provides several methods to interact or configure it further:

- `activateGizmos(bool)`: If `true`, it highlights the gizmos
- `copyState()`, `pasteState()`: Allows you to copy and paste the state of the controls to the clipboard in JSON
- `saveState()`, `reset()`: Internally saves the current state and uses `reset()` to apply the saved state
- `dispose()`: Removes all parts of this control from the scene, and cleans up any listeners and animations
- `setGizomsVisible(bool)`: Specifies whether to show or hide the gizmos
- `setTbRadius(radiusFactor)`: Updates the `radiusFactor` property and redraws the gizmos
- `setMouseAction(operation, mouse, key)`: Determines which mouse key provides which action
- `unsetMouseAction(mouse, key)`: Clears an assigned mouse action
- `update()`: Whenever the camera properties change, call this to apply these new settings to this control
- `getRayCaster()`: Provides access to `rayCaster`, which is used internally by these controls

`ArcballControls` is a really useful and relatively new addition to Three.js that provides advanced control of the scene using the mouse. If you're looking for a simpler approach, you can use `TrackBallControls`.

TrackBallControls

Using `TrackBallControls` follows the same approach as we saw for `ArcballControls`:

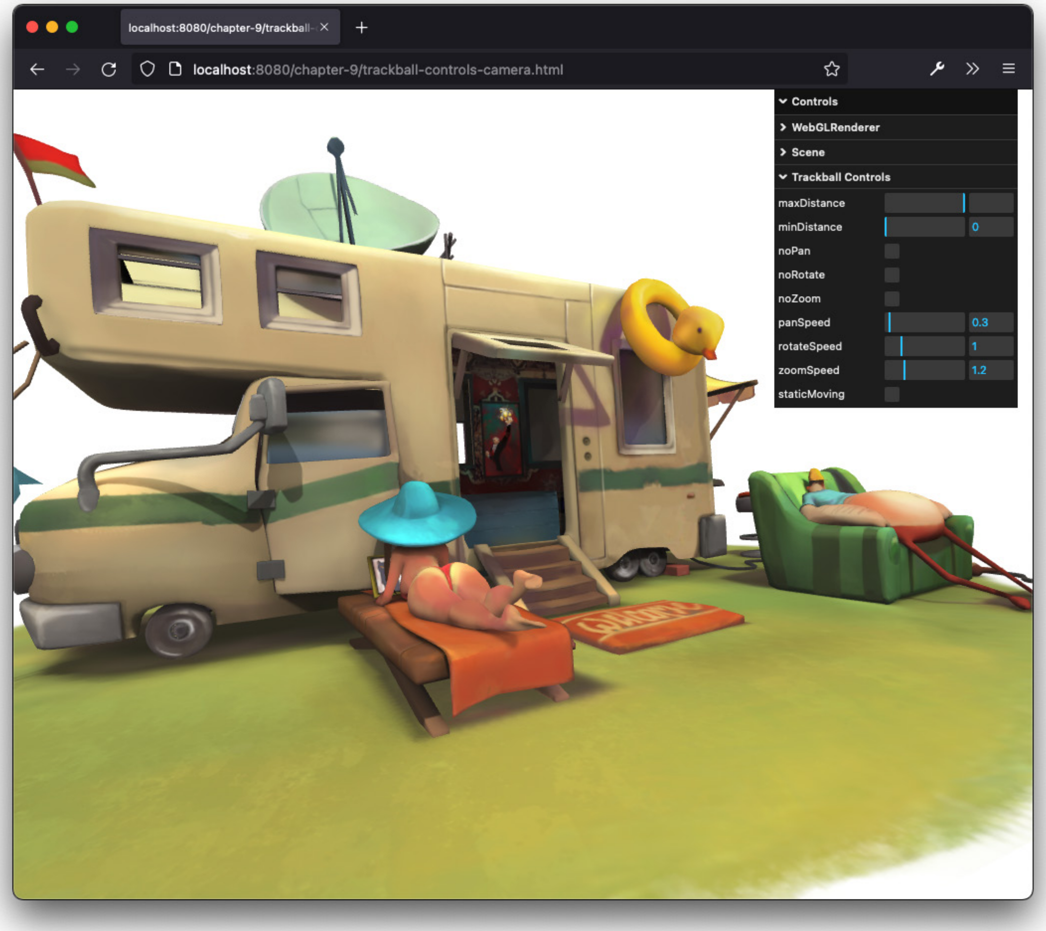
```
import { TrackBallControls } from 'three/examples/jsm/
  controls/TrackBallControls'
const controls = new TrackBallControls(camera, renderer.
  domElement)
```

This time, we just need to pass in the `camera` and `domeElement` properties from the renderer. For the trackball controls to work, we also need to add a `THREE.Clock` and update the render loop, like so:

```
const clock = new THREE.Clock()
function animate() {
  requestAnimationFrame(animate)
  renderer.render(scene, camera)
  controls.update(clock.getDelta())
}
```

In the preceding code snippet, we can see a new Three.js object, `THREE.Clock`. The `THREE.Clock` object can be used to calculate the elapsed time that a specific invocation or rendering loop takes to complete. You can do this by calling the `clock.getDelta()` function. This function will return the elapsed time between this call and the previous call to `getDelta()`. To update the position of the camera, we can call the `TrackBallControls.update()` function. In this function, we need to provide the time that has passed since the last time this update function was called. For this, we can use the `getDelta()` function from the `THREE.Clock` object. You might be wondering why we don't just pass in the frame rate (1/60 seconds) to the update function. The reason is that with `requestAnimationFrame`, we can expect 60 FPS, but this isn't guaranteed. Depending on all kinds of external factors, the frame rate might change. To make sure the camera turns and rotates smoothly, we need to pass in the exact elapsed time.

A working example of this can be found in `trackball-controls-camera.html`. The following screenshot shows a still image of this example:


localhost:8080/chapter-9/trackball-
localhost:8080/chapter-9/trackball-controls-camera.html
Controls
WebGLRenderer
Scene
Trackball Controls

maxDistance
minDistance 0
noPan ☐
noRotate ☐
noZoom ☐
panSpeed 0.3
rotateSpeed 1
zoomSpeed 1.2
staticMoving ☐

Figure 9.10 – Using TrackBallControls to control a scene

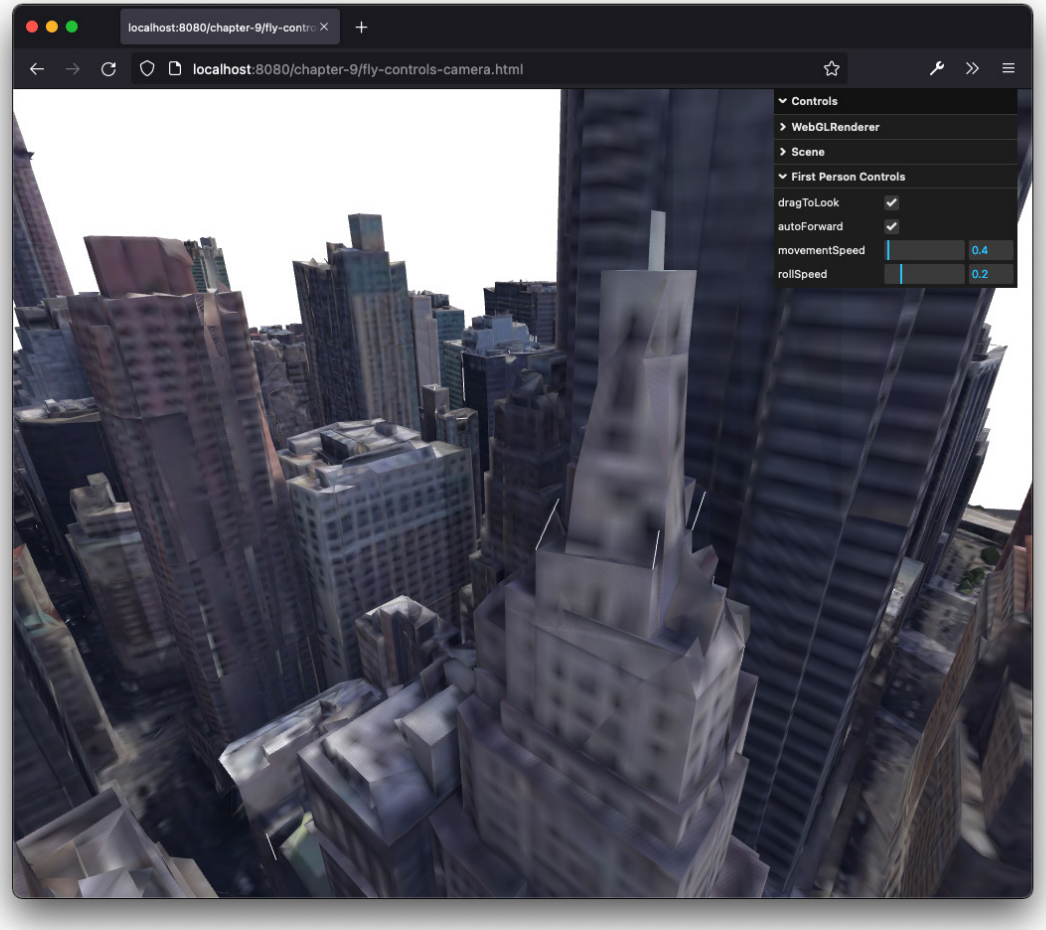
You can control the camera in the following manner:

- **Left mouse button and move**: Rotate and roll the camera around the scene
- **Scroll wheel**: Zoom in and zoom out
- **Middle mouse button and move**: Zoom in and zoom out
- **Right mouse button and move**: Pan around the scene

There are a couple of properties that you can use to fine-tune how the camera acts. For instance, you can set how fast the camera rotates with the `rotateSpeed` property and disable zooming by setting the `noZoom` property to `true`. In this chapter, we won't go into detail on what each property does as they are pretty much self-explanatory. For a complete overview of what is possible, look at the source of the `TrackBallControls.js` file, where these properties are listed.

FlyControls

The next control we'll look at is `FlyControls`. With `FlyControls`, you can fly around a scene using controls also found in flight simulators. An example can be found in `fly-controls-camera.html`. The following screenshot shows a still image of this example:


localhost:8080/chapter-9/fly-contro
localhost:8080/chapter-9/fly-controls-camera.html
Controls
WebGLRenderer
Scene
First Person Controls

dragToLook ✓
autoForward ✓
movementSpeed 0.4
rollSpeed 0.2

Figure 9.11 – Using FlyControls to fly around a scene

Enabling `FlyControls` works in the same manner as the other controls:

```
import { FlyControls } from 'three/examples/jsm/controls/
```

```
FlyControls'
const controls = new FlyControls(camera, renderer.domElement)
const clock = new THREE.Clock()
function animate() {
  requestAnimationFrame(animate)
  renderer.render(scene, camera)
  controls.update(clock.getDelta())
}
```

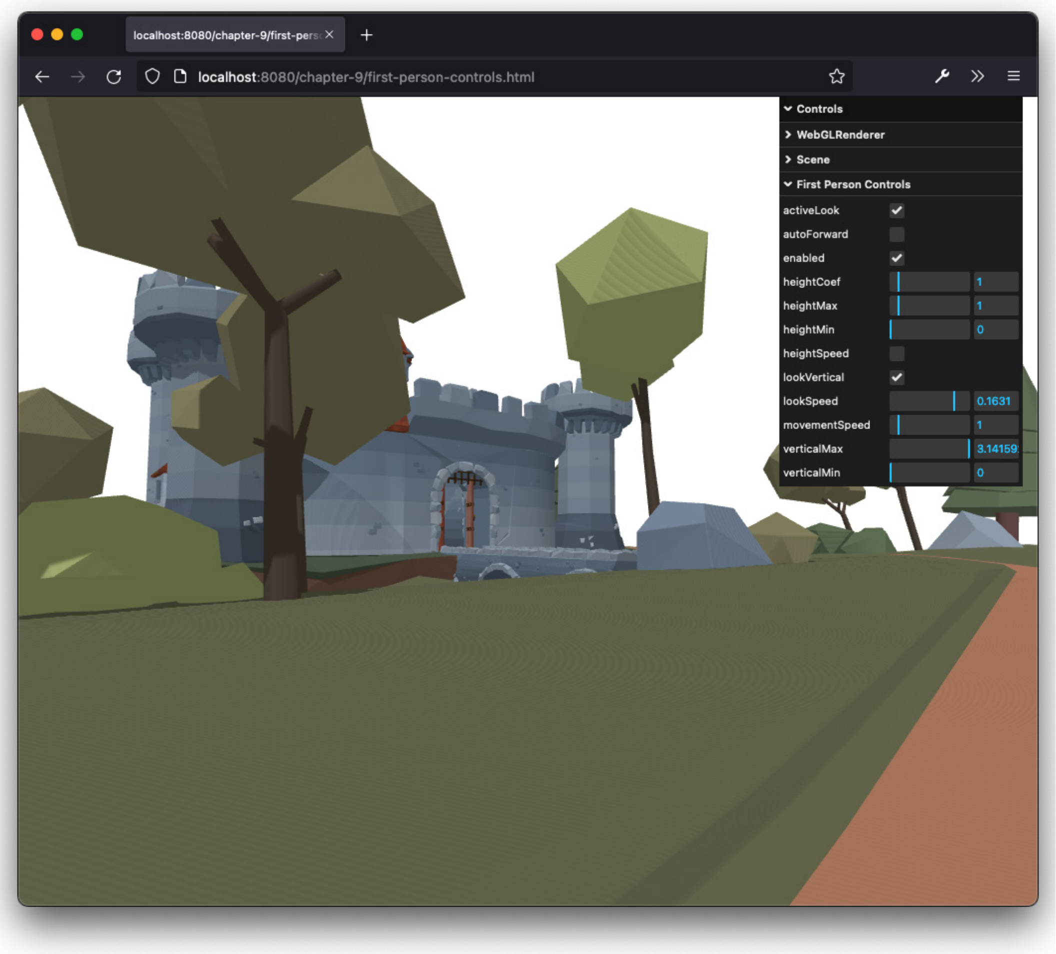
`FlyControls` takes the camera and the renderer's `domElement` as arguments and requires that you call the `update()` function with the elapsed time in the render loop. You can control the camera with `THREE.FlyControls` in the following manner:

- **Left and middle mouse buttons**: Start moving forward
- **Right mouse button**: Move backward
- **Mouse movement**: Look around
- *W*: Start moving forward
- *S*: Move backward
- *A*: Move left
- *D*: Move right
- *R*: Move up
- *F*: Move down
- **Left, right, up, and down arrows**: Look left, right, up, and down, respectively
- *G*: Roll left
- *E*: Roll right

The next control we'll look at is `THREE.FirstPersonControls`.

FirstPersonControls

As the name implies, `FirstPersonControls` allows you to control the camera just like in a first-person shooter. The mouse is used to look around, and the keyboard is used to walk around. You can find an example in `07-first-person-camera.html`. The following screenshot shows a still image of this example:


localhost:8080/chapter-9/first-pers
localhost:8080/chapter-9/first-person-controls.html
Controls
WebGLRenderer
Scene
First Person Controls

activeLook ☑
autoForward ☐
enabled ☑
heightCoef 1
heightMax 1
heightMin 0
heightSpeed ☐
lookVertical ☑
lookSpeed 0.1631
movementSpeed 1
verticalMax 3.14159
verticalMin 0

Figure 9.12 – Exploring a scene using the first-person controls

Creating these controls follows the same principles as the ones followed for the other controls we've seen so far:

```
Import { FirstPersonControls } from 'three/examples/jsm/
  controls/FirstPersonControls'
const controls = new FirstPersonControls(camera, renderer.
domElement)
const clock = new THREE.Clock()
function animate() {
  requestAnimationFrame(animate)
  renderer.render(scene, camera)
  controls.update(clock.getDelta())
}
```

The functionality provided by this control is pretty straightforward:

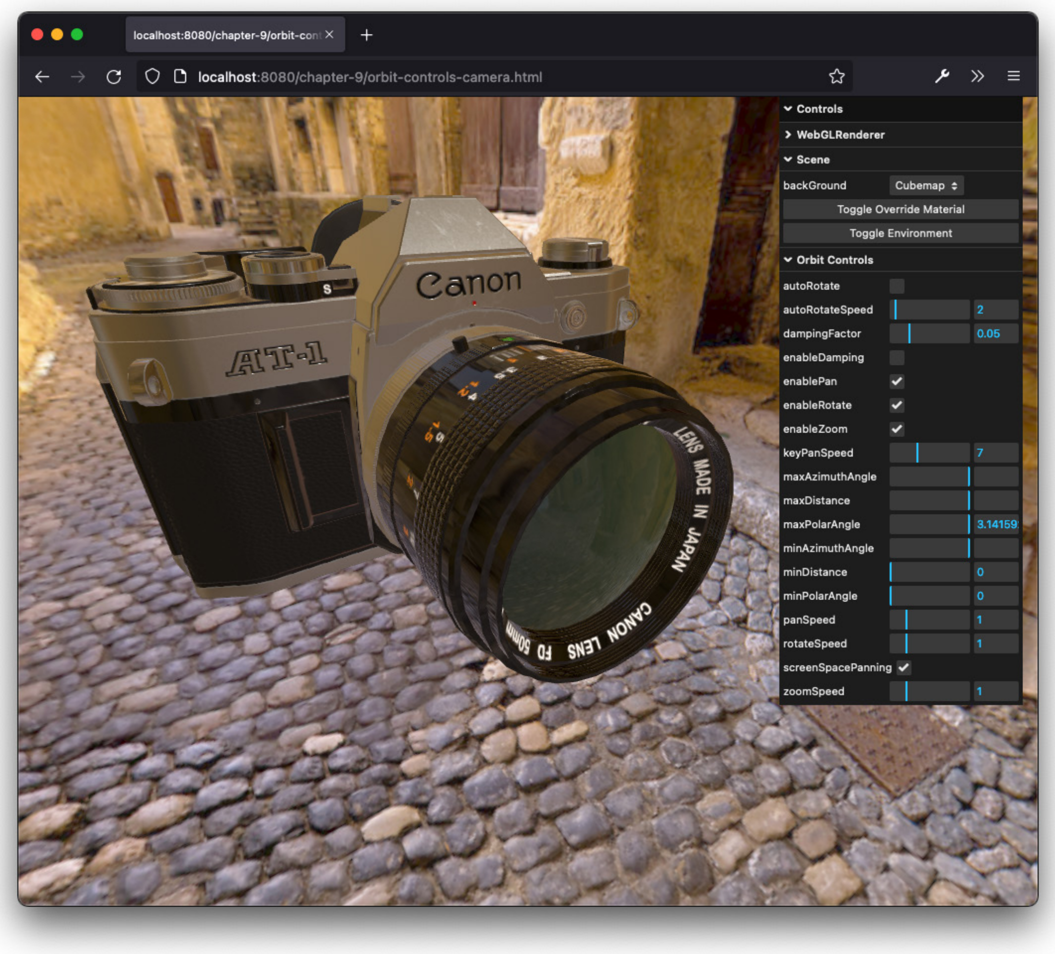
- **Mouse movement**: Look around
- **Left, right, up, and down arrows**: Move left, right, forward, and backward, respectively
- *W*: Move forward
- *A*: Move left
- *S*: Move backward
- *D*: Move right
- *R*: Move up
- *F*: Move down
- *Q*: Stop all movement

For the final control, we'll move on from this first-person perspective to the perspective from space.

OrbitControls

The `OrbitControls` control is a great way to rotate and pan around an object in the center of the scene. This is also the control we used in the other chapters to provide you with a simple way to explore the models in the examples provided.

With `orbit-controls-orbit-camera.html`, we've included an example that shows how this control works. The following screenshot shows a still image of this example:


localhost:8080/chapter-9/orbit-con
localhost:8080/chapter-9/orbit-controls-camera.html
Canon
AT-1
LENS MADE IN JAPAN
CANON LENS FD 50mm
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Orbit Controls

backGround Cubemap
autoRotate ☐
autoRotateSpeed 2
dampingFactor 0.05
enableDamping ☐
enablePan ☑
enableRotate ☑
enableZoom ☑
keyPanSpeed 7
maxAzimuthAngle
maxDistance
maxPolarAngle 3.14159
minAzimuthAngle
minDistance 0
minPolarAngle 0
panSpeed 1
rotateSpeed 1
screenSpacePanning ☑
zoomSpeed 1

Figure 9.13 – OrbitControls properties

Using `OrbitControls` is just as simple as using the other controls. Include the correct JavaScript file, set up the control with the camera, and use `THREE.Clock` again to update the control:

```
import { OrbitControls } from 'three/examples/jsm/
  controls/OrbitControls'
const controls = new OrbitControls(camera, renderer.
  domElement)
const clock = new THREE.Clock()
function animate() {
  requestAnimationFrame(animate)
  renderer.render(scene, camera)
  controls.update(clock.getDelta())
}
```

The controls for `OrbitControls` are focused on using the mouse, as shown in the following list:

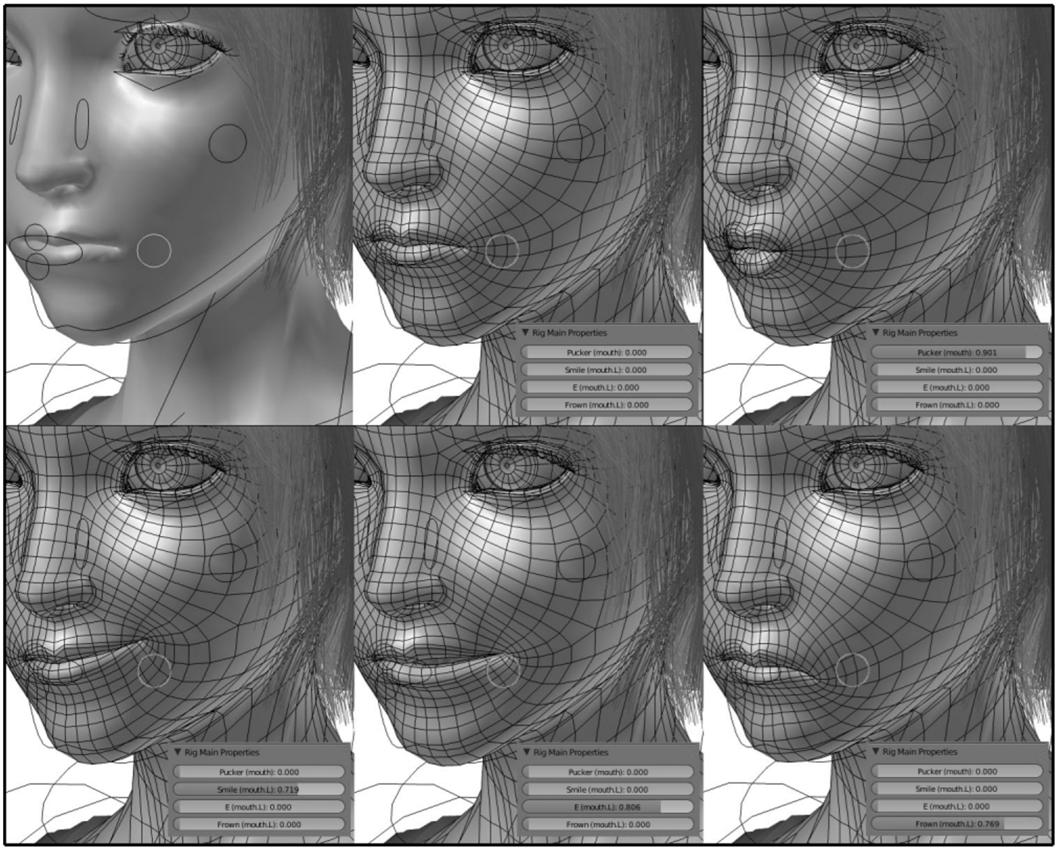
- **Left mouse click and move**: Rotate the camera around the center of the scene
- **Scroll wheel or middle mouse click and move**: Zoom in and zoom out
- **Right mouse click and move**: Pan around the scene

That's it for the camera and moving it around. In this section, we saw a lot of controls that allow you to easily interact with and move through a scene by changing the camera properties. In the next section, we'll look at more advanced methods of animation: morphing and skinning.

Morphing and skeleton animation

When you create animations in external programs (for instance, Blender), you usually have two main options to define animations:

- **Morph targets**: With morph targets, you define a deformed version – that is, a key position – of the mesh. For this deformed target, all vertex positions are stored. All you need to do to animate the shape is move all the vertices from one position to another key position and repeat that process. The following screenshot shows various morph targets used to show facial expressions (this screenshot has been provided by the Blender foundation):



Pucker (mouth): 0.000
Smile (mouth.L): 0.000
E (mouth.L): 0.000
Frown (mouth.L): 0.000

Pucker (mouth): 0.901
Smile (mouth.L): 0.000
E (mouth.L): 0.000
Frown (mouth.L): 0.000

Pucker (mouth): 0.000
Smile (mouth.L): 0.719
E (mouth.L): 0.000
Frown (mouth.L): 0.000

Pucker (mouth): 0.000
Smile (mouth.L): 0.000
E (mouth.L): 0.806
Frown (mouth.L): 0.000

Pucker (mouth): 0.000
Smile (mouth.L): 0.000
E (mouth.L): 0.000
Frown (mouth.L): 0.769

Figure 9.14 – Setting up animations using morph targets

- **Skeleton animation**: An alternative is using skeleton animation. With skeletal animation, you define the skeleton – that is, the bones – of the mesh and attach vertices to the specific bones. Now, when you move a bone, any connected bone is also moved appropriately, and the attached vertices are moved and deformed based on the position, movement, and scaling of the bone. The following screenshot, once again provided by the Blender foundation, shows an example of how bones can be used to move and deform an object:

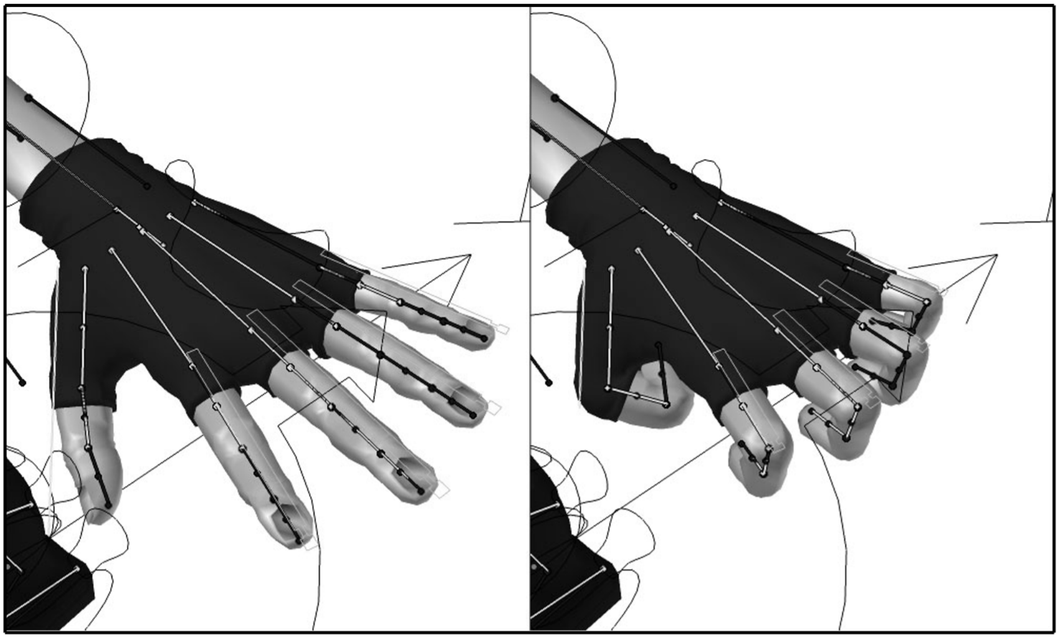


Figure 9.15 – Setting up animations using bones

Three.js supports both modes, but there can be an issue getting a good export when you want to work with skeleton/bones-based animations. For the best results, you should export or convert your model into glTF format, which is becoming the default for exchanging models, animations, and scenes, and has great support from Three.js.

In this section, we'll look at both options and also look at a couple of external formats supported by Three.js in which animations can be defined.

Animation with morph targets

Morph targets are the most straightforward way of defining an animation. You define all the vertices for each important position (also called keyframes) and tell Three.js to move the vertices from one position to the other.

We'll show you how to work with morph targets using two examples. In the first example, we'll let Three.js handle the transition between the various keyframes (or morph targets, as we'll call them from now on), and in the second one, we'll do this manually. Keep in mind that we are only scratching the surface of what is possible with animations in Three.js. As you'll see in this section, Three.js has excellent support for controlling animations, supports syncing of animations, and provides ways to smoothly transition from one animation to another, warranting a book just on this subject. So, in the next couple of sections, we'll provide you with the basics of animations in Three.js, which should provide you with enough information to get started and explore the more complex subjects.

Animation with a mixer and morph targets

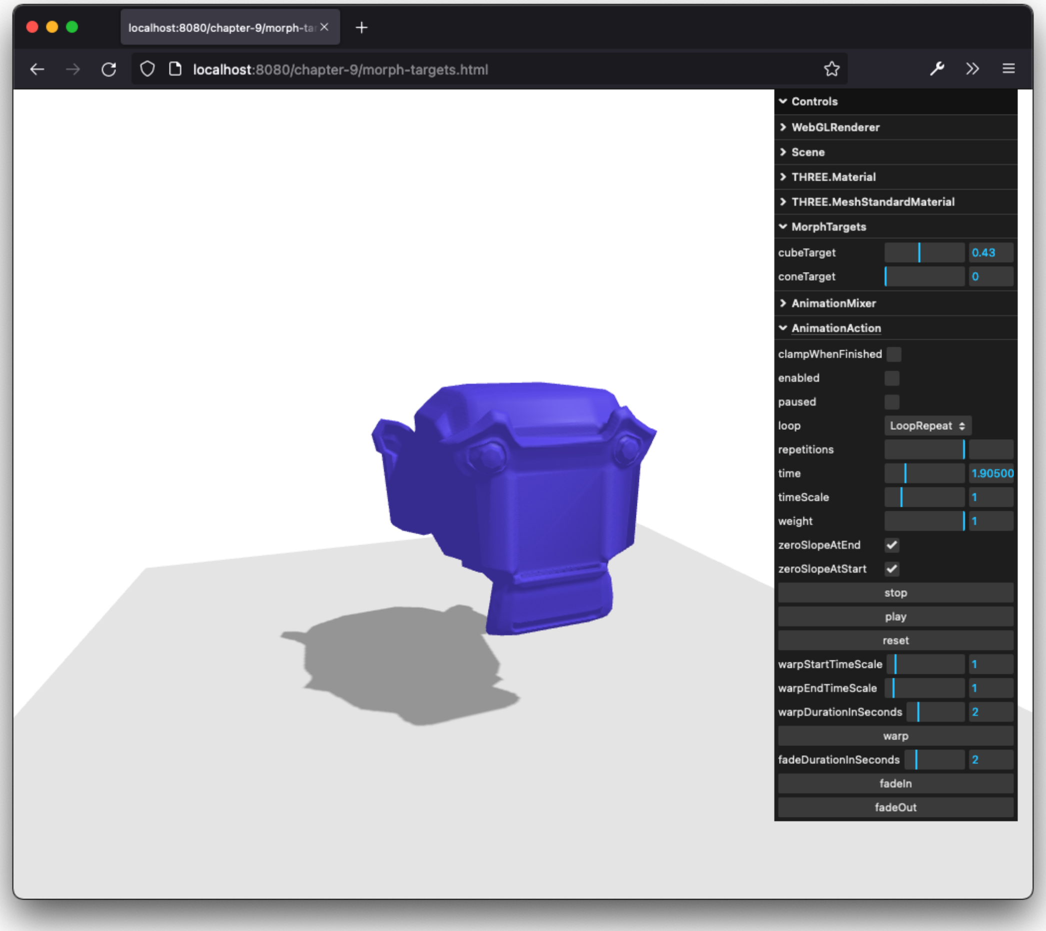
Before we dive into the examples, first, we'll look at the three core classes that you can use to animate with Three.js. Later in this chapter, we'll show you all the functions and properties provided by these objects:

- `THREE.AnimationClip`: When you load a model that contains animations, you can look in the `response` object for a field usually called `animations`. This field will contain a list of `THREE.AnimationClip` objects. Note that depending on the loader, an animation might be defined on a `Mesh`, a `Scene`, or be provided completely separately. A `THREE.AnimationClip` most often holds the data for a certain animation the model you loaded can perform. For instance, if you loaded a model of a bird, one `THREE.AnimationClip` would contain the information needed to flap the wings, and another one might be opening and closing its beak.
- `THREE.AnimationMixer`: `THREE.AnimationMixer` is used to control several `THREE.AnimationClip` objects. It makes sure the timing of the animation is correct and makes it possible to sync animations together, or cleanly move from one animation to another.
- `THREE.AnimationAction`: `THREE.AnimationMixer` itself doesn't expose a large number of functions to control the animation, though. This is done through `THREE.AnimationAction` objects, which are returned when you add a `THREE.AnimationClip` to a `THREE.AnimationMixer` (though you can get them at a later time by using functions provided by `THREE.AnimationMixer`).

There is also an `AnimationObjectGroup`, which you can use to provide the animation state not just to a single `Mesh` but to a group of objects.

In the following example, you can control a `THREE.AnimationMixer` and a `THREE.AnimationAction`, which were created using a `THREE.AnimationClip` from the model. The `THREE.AnimationClip` objects used in this example morph a model into a cube and then into a cylinder.

For this first morphing example, the easiest way to understand how a morph targets-based animation works is by opening up the `morph-targets.html` example. The following screenshot shows a still image of this example:


localhost:8080/chapter-9/morph-ta
localhost:8080/chapter-9/morph-targets.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
MorphTargets
AnimationMixer
AnimationAction
stop
play
reset
warp
fadeIn
fadeOut

cubeTarget 0.43
coneTarget 0
clampWhenFinished ☐
enabled ☐
paused ☐
loop LoopRepeat
repetitions
time 1.90500
timeScale 1
weight 1
zeroSlopeAtEnd ☑
zeroSlopeAtStart ☑
warpStartTimeScale 1
warpEndTimeScale 1
warpDurationInSeconds 2
fadeDurationInSeconds 2

Figure 9.16 – Animation using morph targets

In this example, we've got a simple model (a monkey's head), which can be transformed into either a cube or a cylinder using morph targets. You can easily test this yourself by moving the `cubeTarget` or `coneTarget` sliders, and you'll see the head being morphed into a different shape. For instance, with `cubeTarget` at `0.5`, you will see that we're halfway through morphing the monkey's initial head into a cube. Once it is at `1`, the initial geometry is morphed completely:

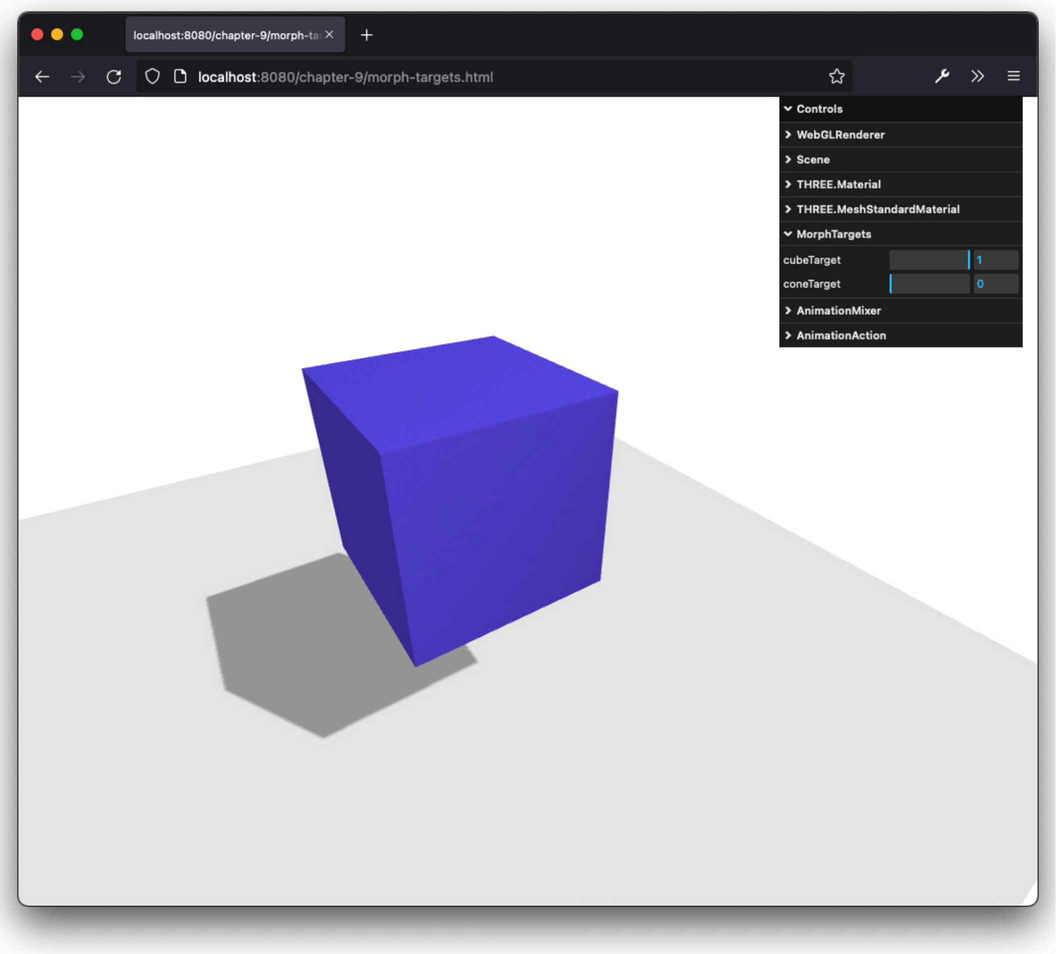

localhost:8080/chapter-9/morph-ta
localhost:8080/chapter-9/morph-targets.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
MorphTargets
cubeTarget
1
coneTarget
0
AnimationMixer
AnimationAction

Figure 9.17 – Same model, but now with cubeTarget set to 1

And that's the basics of how morph animations work. You've got several `morphTargets` (influences) you can control, and based on their value (from 0 to 1), the vertices move into the desired position. An animation that uses morph targets uses this approach. It just defines at which time certain vertex positions should occur. When running the animation, Three.js will make sure the right values are passed to the `morphTargets` property of the `Mesh` instance.

To run the predefined animation, you can open the `AnimationMixer` menu for this example, and click **Play**. You'll see that the head will first transform into a cube and then into a cylinder, before moving back into the shape of a head.

Setting up the required components to accomplish this in Three.js can be done using the following pieces of code. First, we have to load the model. In this example, we exported this example from

Blender into glTF, so our `animations` are at the top level. We simply add these to a variable that we can access in the other parts of the code. We could also set this as a property on the mesh or add it to the `userdata` property of `Mesh`:

```
let animations = []
const loadModel = () => {
  const loader = new GLTFLoader()
  return loader.loadAsync('/assets/models/blender-morph-
targets/morph-targets.gltf').then((container) => {
    animations = container.animations
    return container.scene
  })
}
```

Now that we've got an animation from the loaded model, we can set up the specific Three.js components so that we can play them:

```
const mixer = new THREE.AnimationMixer(mesh)
const action = mixer.clipAction(animations[0])
action.play()
```

There is one final step we need to take so that the correct shape of the mesh is shown whenever we render something, and that is adding a single line to the render loop:

```
// in render loop
mixer.update(clock.getDelta())
```

Here, we used `THREE.Clock` again to determine the time that's passed between now and the previous render loop, and called `mixer.update()`. This information is used by the mixer to determine how far it should morph the vertices to the next morph target (keyframe).

`THREE.AnimationMixer` and `THREE.AnimationClip` provide several other functions that you can use to control the animation or create new `THREE.AnimationClip` objects. You can experiment with them by using the menu on the right in the examples for this section. We will start with `THREE.AnimationClip`:

- `duration`: The duration of this track (in seconds).
- `name`: The name of this clip.

- `tracks`: The internal property used to keep track of how certain properties of the model are animated.
- `uuid`: The unique ID of this clip. This is assigned automatically.
- `clone()`: Makes a copy of this clip.
- `optimize()`: This optimizes `THREE.AnimationClip`.
- `resetDuration()`: This determines the correct duration of this clip.
- `toJson()`: Converts this clip into a JSON object.
- `trim()`: This trims all the internal tracks to the duration set on this clip.
- `validate()`: Does some minimal validation to see if this is a valid clip.
- `CreateClipsFromMorphTargetSequences( name, morphTargetSequences,fps, noLoop)`: This creates a list of `THREE.AnimationClip` instances based on a set of morph target sequences.
- `CreateFromMorpTargetSequences( name, morphTargetSequence,fps,noLoop)`: This creates a single `THREE.AnimationClip` from a sequence of morph targets.
- `findByName(objectOrClipArray, name)`: Searches for a `THREE.AnimationClip` by name.
- `parse` and `toJson`: Allow you to restore and save a `Three.AnimationClip` as JSON, respectively.
- `parseAnimation(animation, bones)`: Converts an `THREE.AnimationClip` into JSON.

Once you've got a `THREE.AnimationClip`, you can pass it into the `THREE.AnimationMixer` object, which provides the following functionality:

- `AnimationMixer(rootObject)`: The constructor for this object. This constructor takes a `THREE.Object3D` as an argument (for example, a `THREE.Mesh` of a `THREE.Group`).
- `time`: The global time for this mixer. This starts at 0, at the time when this mixer is created.
- `timeScale`: This can be used to speed up or slow down all the animations managed by this mixer. If the value of this property is set to 0, all the animations are effectively paused.
- `clipAction(animationClip, optionalRoot)`: This creates a `THREE.AnimationAction` that can be used to control the passed-in `THREE.AnimationClip`. If the animation clip is for a different object than what was provided in the constructor of `AnimationMixer`, you can pass that in as well.

- `existingAction(animationClip, optionalRoot)`: This returns the `THREE.AnimationAction` property, which can be used to control the passed-in `THREE.AnimationClip`. Once again, if `THREE.AnimationClip` is for a different `rootObject`, you can also pass that in.

When you get `THREE.AnimationClip` back, you can use it to control the animation:

- `clampWhenFinished`: When set to `true`, this will cause the animation to be paused when it reaches its last frame. The default is `false`.
- `enabled`: When set to `false`, this will disable the current action so that it does not affect the model. When the action is re-enabled, the animation will continue where it left off.
- `loop`: This is the looping mode of this action (which can be set using the `setLoop` function). This can be set to the following:
    - `THREE.LoopOnce`: Plays the clip only one time
    - `THREE.LoopRepeat`: Repeats the clip based on the number of repetitions that have been set
    - `THREE.LoopPingPong`: Plays the clip based on the number of repetitions, but alternates between playing the clip forward and backward
- `paused`: Setting this property to `true` will pause the execution of this clip.
- `repetitions`: The number of times the animation will be repeated. This is used by the `loop` property. The default is `Infinity`.
- `time`: The time this action has been running. This is wrapped from 0 to the duration of the clip.
- `timeScale`: This can be used to speed up or slow down this animation. If the value of this property is set to `0`, this animation is effectively paused.
- `weight`: This specifies the effect this animation has on the model from a scale of `0` to `1`. When set to `0`, you won't see any transformation of the model from this animation, and when set to `1`, you will see the full effect of this animation.
- `zeroSlopeAtEnd`: When set to true (which is the default), this will make sure there is a smooth transition between separate clips.
- `zeroSlopeAtStart`: When set to true (which is the default), this will make sure there is a smooth transition between separate clips.
- `crossFadeFrom(fadeOutAction, durationInSeconds, warpBoolean)`: This causes this action to fade in, while `fadeOutAction` is faded out. The total fade takes `durationInSeconds`. This allows for smooth transitions between animations. When `warpBoolean` is set to `true`, it will apply additional smoothing of timescales.
- `crossFadeTo(fadeInAction, durationInSeconds, warpBoolean)`: Same as `crossFadeFrom`, but this time, it fades in the provided action, and fades out this action.

- `fadeIn(durationInSeconds)`: Increases the `weight` property slowly from `0` to `1` within the passed time interval.
- `fadeOut(durationInSeconds)`: Decreases the `weight` property slowly from `0` to `1` within the passed time interval.
- `getEffectiveTimeScale()`: Returns the effective timescale based on the currently running warp.
- `getEffectiveWeight()`: Returns the effective weight based on the currently running fade.
- `getClip()`: Returns the `THREE.AnimationClip` property this action is managing.
- `getMixer()`: Returns the mixer that is playing this action.
- `getRoot()`: Gets the root object that is controlled by this action.
- `halt(durationInSeconds)`: Gradually decreases `timeScale` to `0` within `durationInSeconds`.
- `isRunning()`: Checks whether the animation is currently running.
- `isScheduled()`: Checks whether this action is currently active in the mixer.
- `play()`: Starts running this action (starting the animation).
- `reset()`: Resets this action. This will result in setting `paused` to `false`, `enabled` to `true`, and `time` to `0`.
- `setDuration(durationInSeconds)`: Sets the duration of a single loop. This will change `timeScale` so that the complete animation can play within `durationInSeconds`.
- `setEffectiveTimeScale(timeScale)`: Sets `timeScale` to the provided value.
- `setEffectiveWeight()`: Sets `weight` to the provided value.
- `setLoop(loopMode, repetitions)`: Sets `loopMode` and the number of `repetitions`. See the `loop` property for the options and their effect.
- `startAt(startTimeInSeconds)`: Delays starting the animation for `startTimeInSeconds`.
- `stop()`: Stops this action, and `reset` is applied.
- `stopFading()`: Stops any scheduled fading.
- `stopWarping()`: Stops any schedule warping.
- `syncWith(otherAction)`: Syncs this action with the passed-in action. This will set this action's `time` and `timeScale` value to the passed-in action.
- `warp(startTimeScale, endTimeScale, durationInSeconds)`: Changes the `timeScale` property from `startTimeScale` to `endTimeScale` within the specified `durationInSeconds`.

Besides all the functions and properties that you can use to control the animation, `THREE.AnimationMixer` also provides two events you can listen to by calling `addEventListener` on the mixer. The `"loop"` event is sent when a single loop is finished, and the `"finished"` event is sent when the complete action has finished.

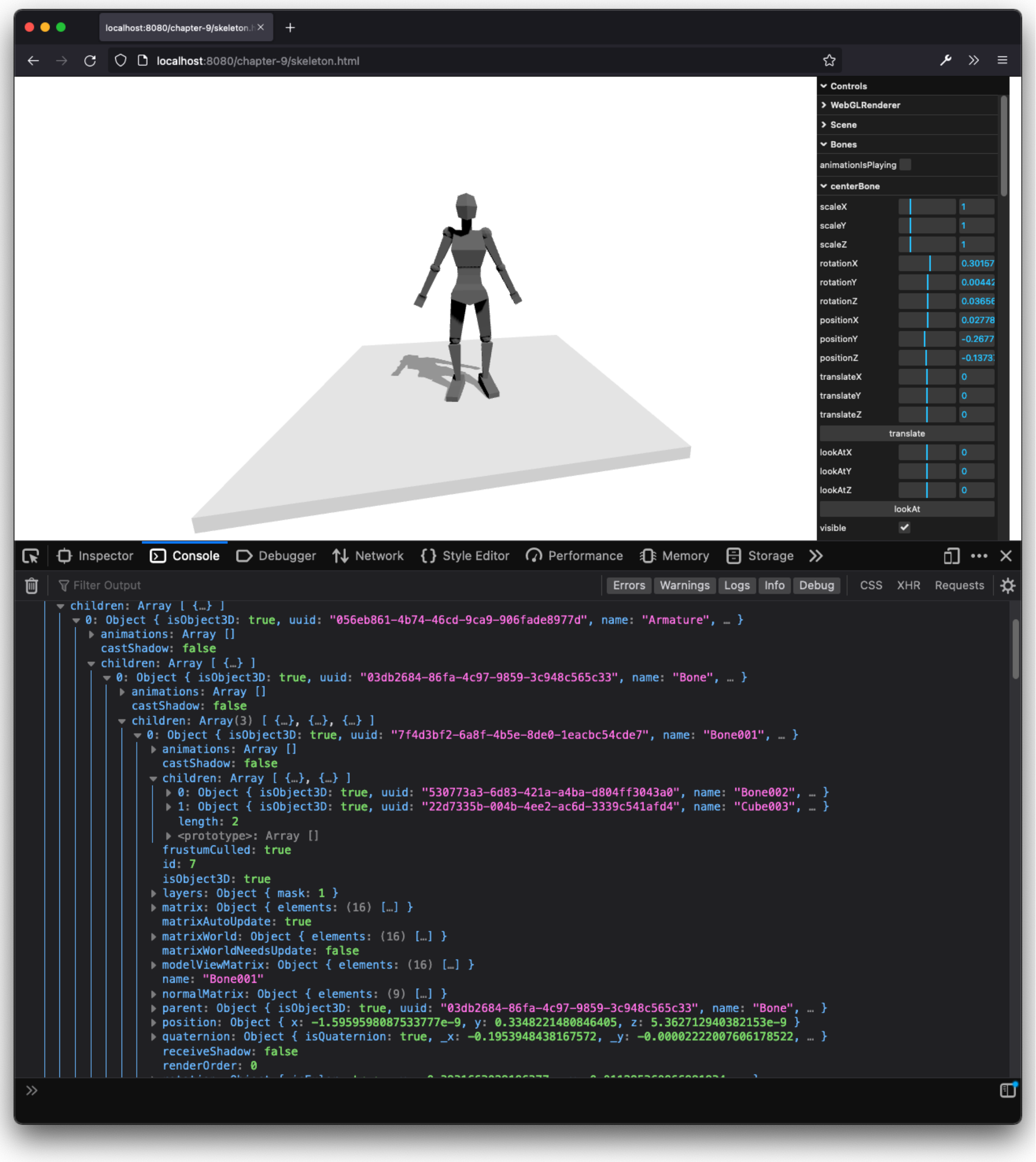
Animation using bones and skinning

As we saw in the *Animation with a mixer and morph targets* section, morph animations are very straightforward. Three.js knows all the target vertex positions and only needs to transition each vertex from one position to the next. For bones and skinning, it becomes a bit more complex. When you use bones for animation, you move the bone, and Three.js has to determine how to translate the attached skin (a set of vertices) accordingly. For this example, we will use a model that was exported from Blender into Three.js format (`lpp-rigging.gltf` in the `models/blender-skeleton` folder). This is a model of a person, complete with a set of bones. By moving the bones around, we can animate the complete model. First, let's look at how we loaded the model:

```
let animations = []
const loadModel = () => {
  const loader = new GLTFLoader()
  return loader.loadAsync('/assets/models/blender-
    skeleton/lpp-rigging.gltf').then((container) => {
    container.scene.translateY(-2)
    applyShadowsAndDepthWrite(container.scene)
    animations = container.animations
    return container.scene
  })
}
```

We've exported the model in glTF format since the support for glTF in Three.js is good. Loading a model for bone animation isn't that different than any of the other models. We just specify the model file and load it like any other glTF file. For glTF, the animations are in a separate property of the object that is loaded, so we simply assign it to the `animations` variable for easy access.

In this example, we've added a console log, which shows what `THREE.Mesh` looks like once we've loaded it:


localhost:8080/chapter-9/skeleton.
localhost:8080/chapter-9/skeleton.html
Controls
WebGLRenderer
Scene
Bones
centerBone
translate
lookAt
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Filter Output
Errors
Warnings
Logs
Info
Debug
CSS
XHR
Requests

animationIsPlaying ☐
scaleX 1
scaleY 1
scaleZ 1
rotationX 0.30157
rotationY 0.00442
rotationZ 0.03656
positionX 0.02778
positionY -0.2677
positionZ -0.1373
translateX 0
translateY 0
translateZ 0
lookAtX 0
lookAtY 0
lookAtZ 0
visible ☑

```
children: Array [ {…} ]
  0: Object { isObject3D: true, uuid: "056eb861-4b74-46cd-9ca9-906fade8977d", name: "Armature", … }
    animations: Array []
    castShadow: false
    children: Array [ {…} ]
      0: Object { isObject3D: true, uuid: "03db2684-86fa-4c97-9859-3c948c565c33", name: "Bone", … }
        animations: Array []
        castShadow: false
        children: Array(3) [ {…}, {…}, {…} ]
          0: Object { isObject3D: true, uuid: "7f4d3bf2-6a8f-4b5e-8de0-1eacbc54cde7", name: "Bone001", … }
            animations: Array []
            castShadow: false
            children: Array [ {…}, {…} ]
              0: Object { isObject3D: true, uuid: "530773a3-6d83-421a-a4ba-d804ff3043a0", name: "Bone002", … }
              1: Object { isObject3D: true, uuid: "22d7335b-004b-4ee2-ac6d-3339c541afd4", name: "Cube003", … }
              length: 2
              <prototype>: Array []
            frustumCulled: true
            id: 7
            isObject3D: true
            layers: Object { mask: 1 }
            matrix: Object { elements: (16) […] }
            matrixAutoUpdate: true
            matrixWorld: Object { elements: (16) […] }
            matrixWorldNeedsUpdate: false
            modelViewMatrix: Object { elements: (16) […] }
            name: "Bone001"
            normalMatrix: Object { elements: (9) […] }
            parent: Object { isObject3D: true, uuid: "03db2684-86fa-4c97-9859-3c948c565c33", name: "Bone", … }
            position: Object { x: -1.5959598087533777e-9, y: 0.3348221480846405, z: 5.362712940382153e-9 }
            quaternion: Object { isQuaternion: true, _x: -0.1953948438167572, _y: -0.00002222007606178522, … }
            receiveShadow: false
            renderOrder: 0
```

Figure 9.18 – The skeleton structure is reflected in the hierarchy of objects

Here, you can see that the mesh consists of a tree of bones and meshes. This also means that if you move a bone, the relevant meshes will be moved alongside it.

The following screenshot shows a still image of this example:

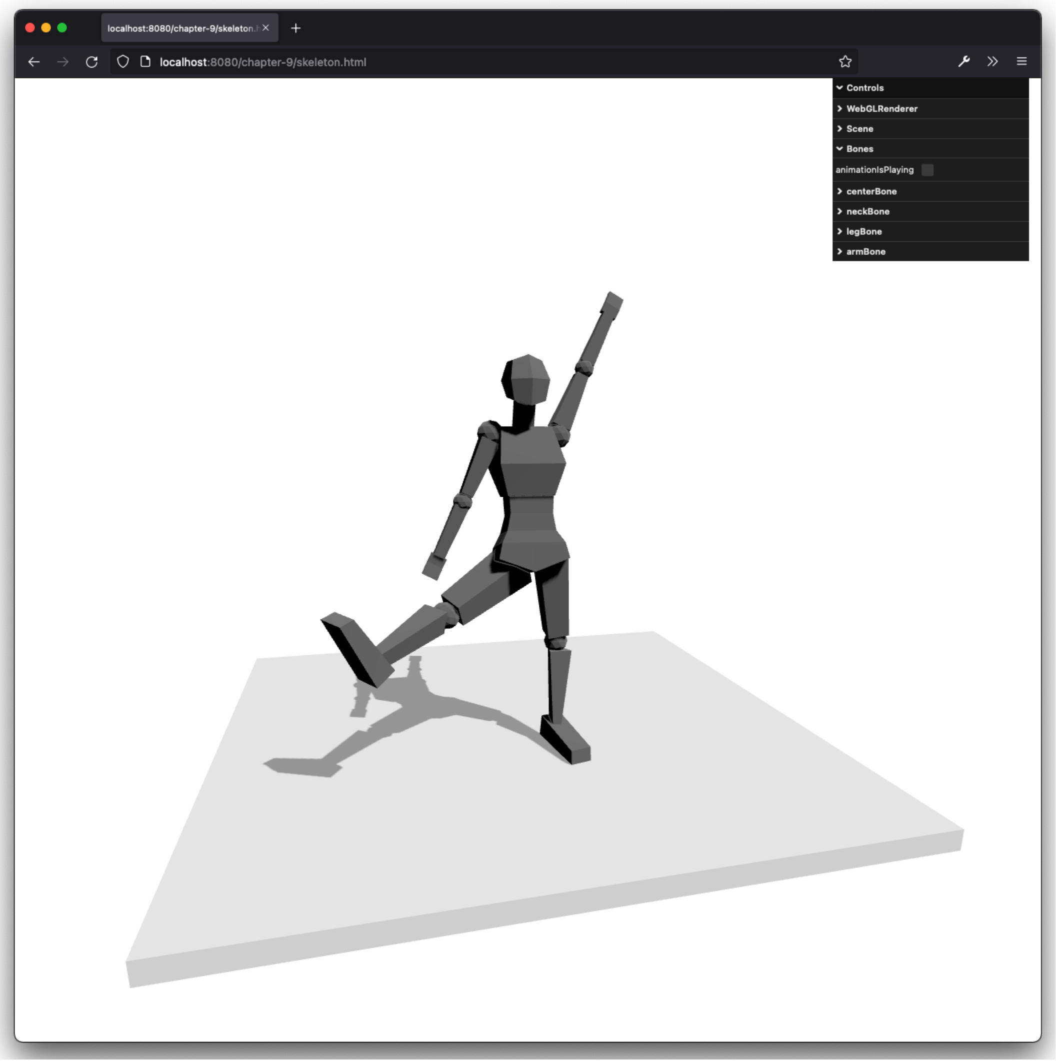

localhost:8080/chapter-9/skeleton.html
Controls
WebGLRenderer
Scene
Bones
animationIsPlaying
centerBone
neckBone
legBone
armBone

Figure 9.19 – Manually changing the rotation of the arm and leg bones

This scene also contains an animation, which you can trigger by checking the **animationIsPlaying** checkbox. This will override the manually set bones' positions and rotations, and has the skeleton kind of jumping up and down:

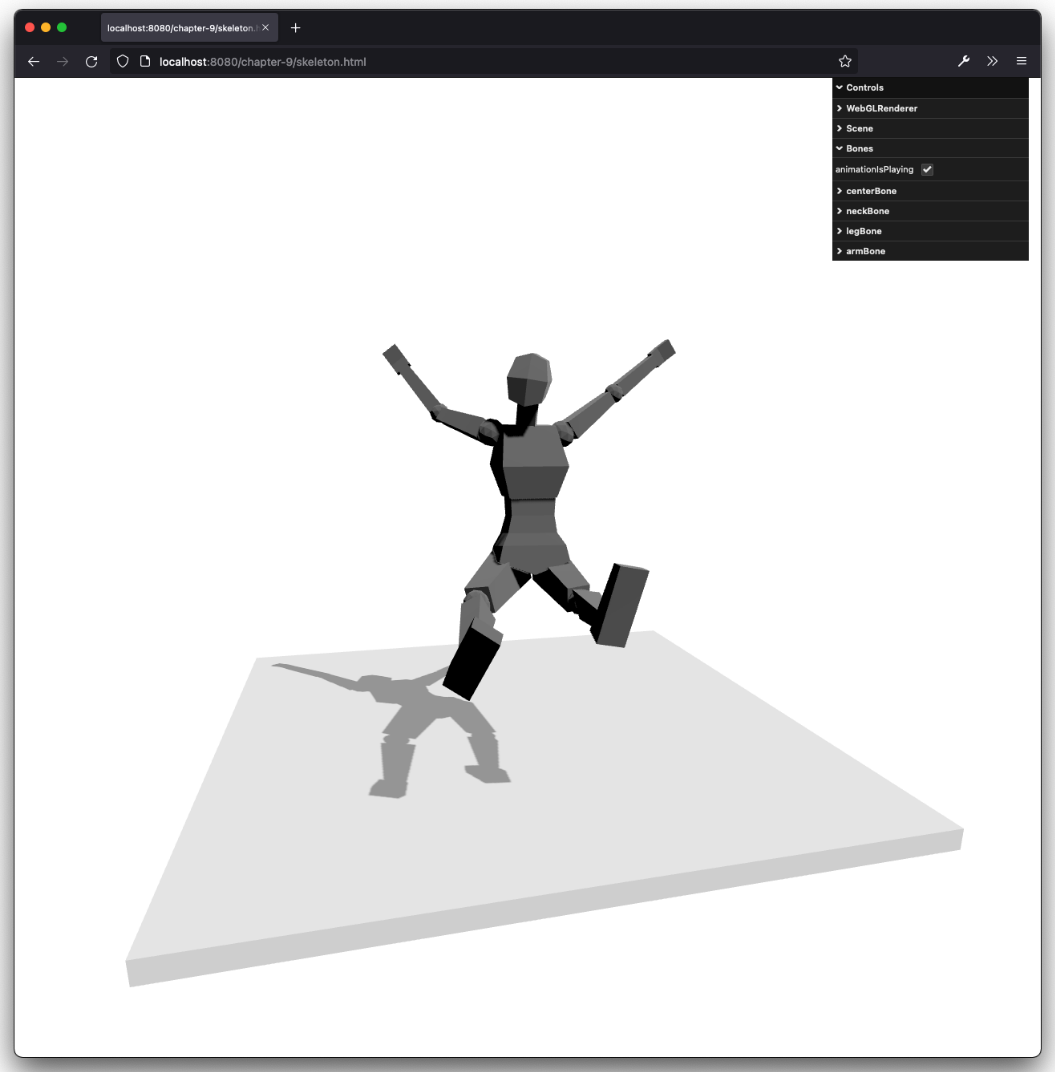

localhost:8080/chapter-9/skeleton.h
localhost:8080/chapter-9/skeleton.html
Controls
WebGLRenderer
Scene
Bones
animationIsPlaying ☑
centerBone
neckBone
legBone
armBone

Figure 9.20 – Playing a skeleton animation

To set up this animation, we must follow the same steps we saw earlier:

```
const mixer = new THREE.AnimationMixer(mesh)
const action = mixer.clipAction(animations[0])
action.play()
```

As you can see, working with bones is just as easy as working with fixed morph targets. In this example, we've only adjusted the rotation of the bones; you can also move the position or change the scale. In the next section, we will look at loading animations from external models.

Creating animations using external models

In *Chapter 8*, *Creating and Loading Advanced Meshes and Geometries*, we looked at several 3D formats that are supported by Three.js. A couple of those formats also support animations. In this chapter, we'll look at the following examples:

- **COLLADA model**: The COLLADA format has support for animations. For this example, we'll load an animation from a COLLADA file and render it with Three.js.
- **MD2 model**: The MD2 model is a simple format used in the older Quake engines. Even though the format is a bit dated, it is still a very good format for storing character animations.
- **glTF models**: **GL transmission format** (**glTF**) is a format specifically designed for storing 3D scenes and models. It focuses on minimizing the size of the assets and tries to be as efficient as possible in unpacking the models.
- **FBX model**: FBX is a format produced by the Mixamo tooling available at `https://www.mixamo.com`. With Mixamo, you can easily rig and animate models, without needing lots of modeling experience.
- **BVH model**: The **Biovision** (**BVH**) format is a slightly different one compared to the other loaders. With this loader, you don't load a geometry with a skeleton or a set of animations. With this format, which is used by Autodesk MotionBuilder, you just load a skeleton, which you can visualize or even attach to your geometry.

We'll start with a glTF model since this format is becoming the standard for exchanging models between different tools and libraries.

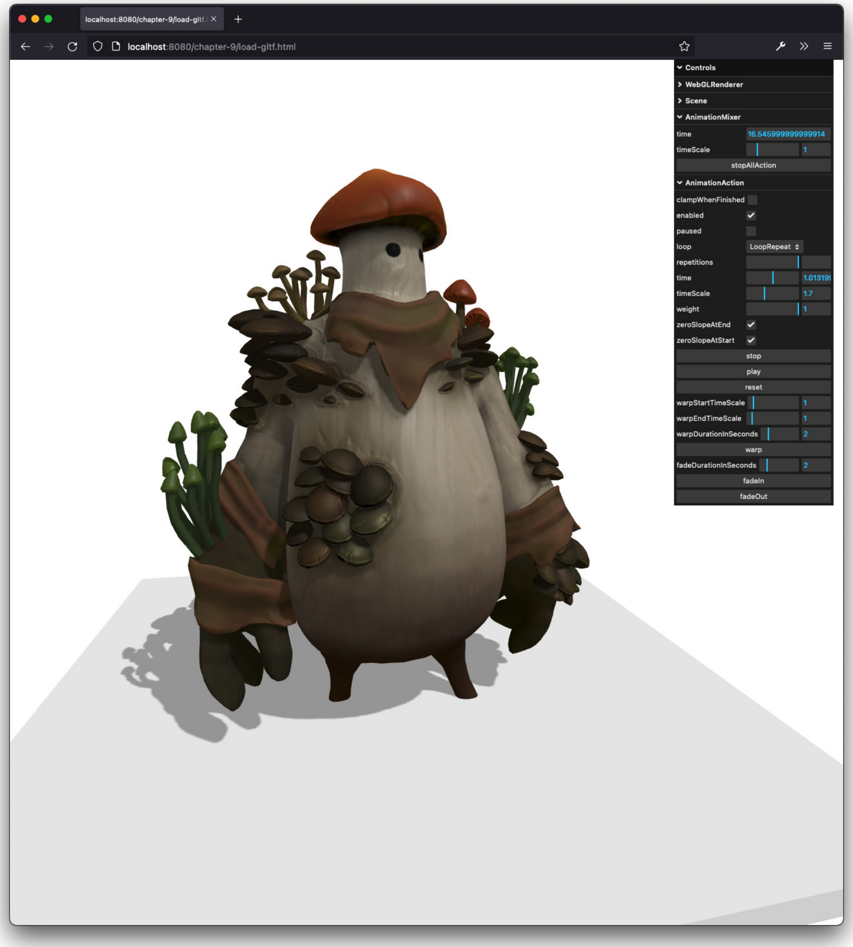
Using gltfLoader

A format that is getting more and more attention lately is the glTF format. This format, for which you can find a very extensive explanation at `https://github.com/KhronosGroup/glTF`, focuses on optimizing size and resource usage. Using `glTFLoader` is similar to using the other loaders:

```
import { GLTFLoader } from 'three/examples
  /jsm/loaders/GLTFLoader'
...
return loader.loadAsync('/assets/models/truffle_man/scene.
gltf').
  then((container) => {
  container.scene.scale.setScalar(4)
```

```
    container.scene.translateY(-2)
    scene.add(container.scene)

    const mixer = new THREE.AnimationMixer( container.scene );
    const animationClip = container.animations[0];
    const clipAction = mixer.clipAction( animationClip ).
play();
})
```

This loader also loads a complete scene, so you can either add everything to the group or select child elements. For this example, you can view the results by opening `load-gltf.js`:


localhost:8080/chapter-9/load-gltf
localhost:8080/chapter-9/load-gltf.html
Controls
WebGLRenderer
Scene
AnimationMixer
stopAllAction
AnimationAction
stop
play
reset
warp
fadeIn
fadeOut

time 16.545999999999914
timeScale 1
clampWhenFinished ☐
enabled ☑
paused ☐
loop LoopRepeat
repetitions
time 1.013199
timeScale 1.7
weight 1
zeroSlopeAtEnd ☑
zeroSlopeAtStart ☑
warpStartTimeScale 1
warpEndTimeScale 1
warpDurationInSeconds 2
fadeDurationInSeconds 2

Figure 9.21 – Animation loaded using glTF

For the next example, we'll use the FBX model.

Visualizing motions captured models using fbxLoader

The Autodesk FBX format has been around for a while and is very easy to use. There is a great resource online where you can find many animations that you can download in this format: `https://www.mixamo.com/`. This site provides 2,500 animations that you can use and customize:

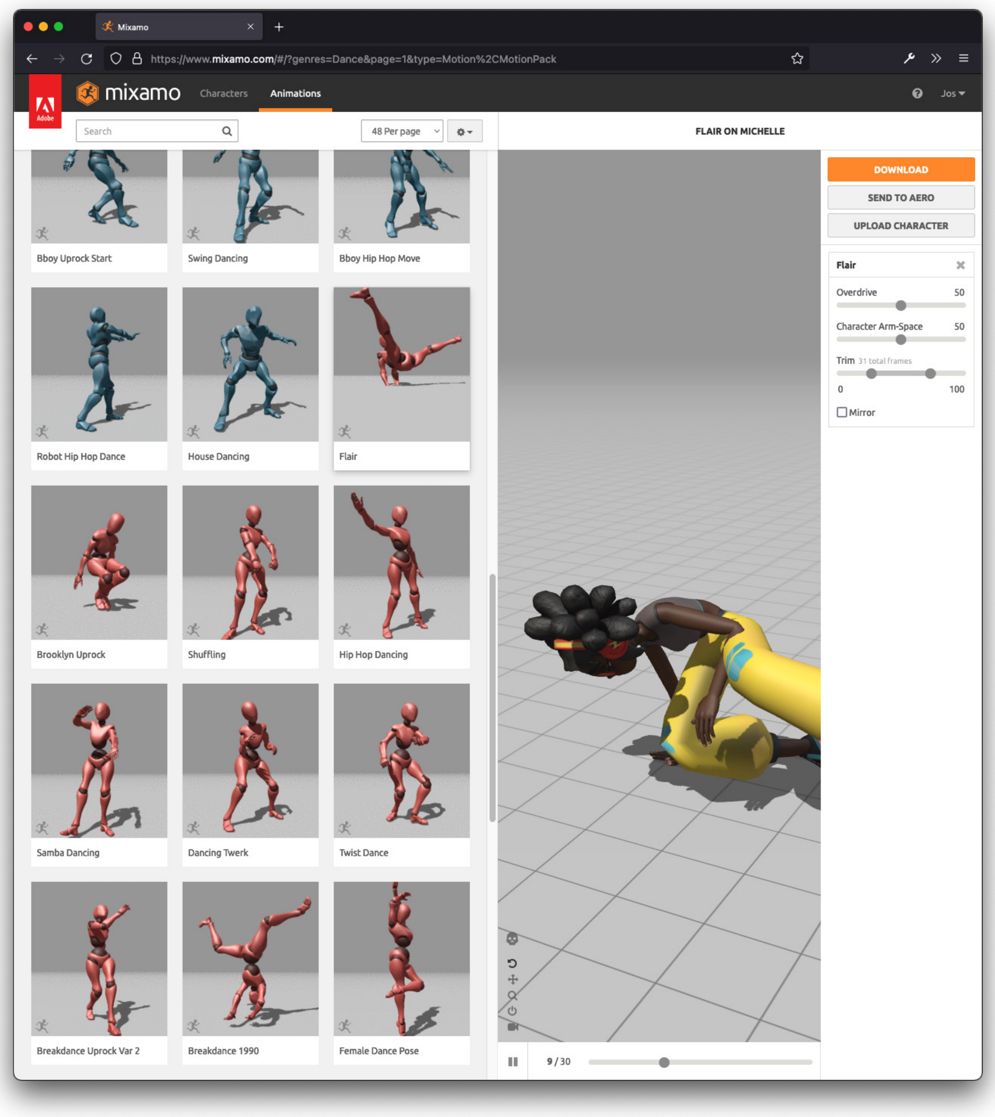
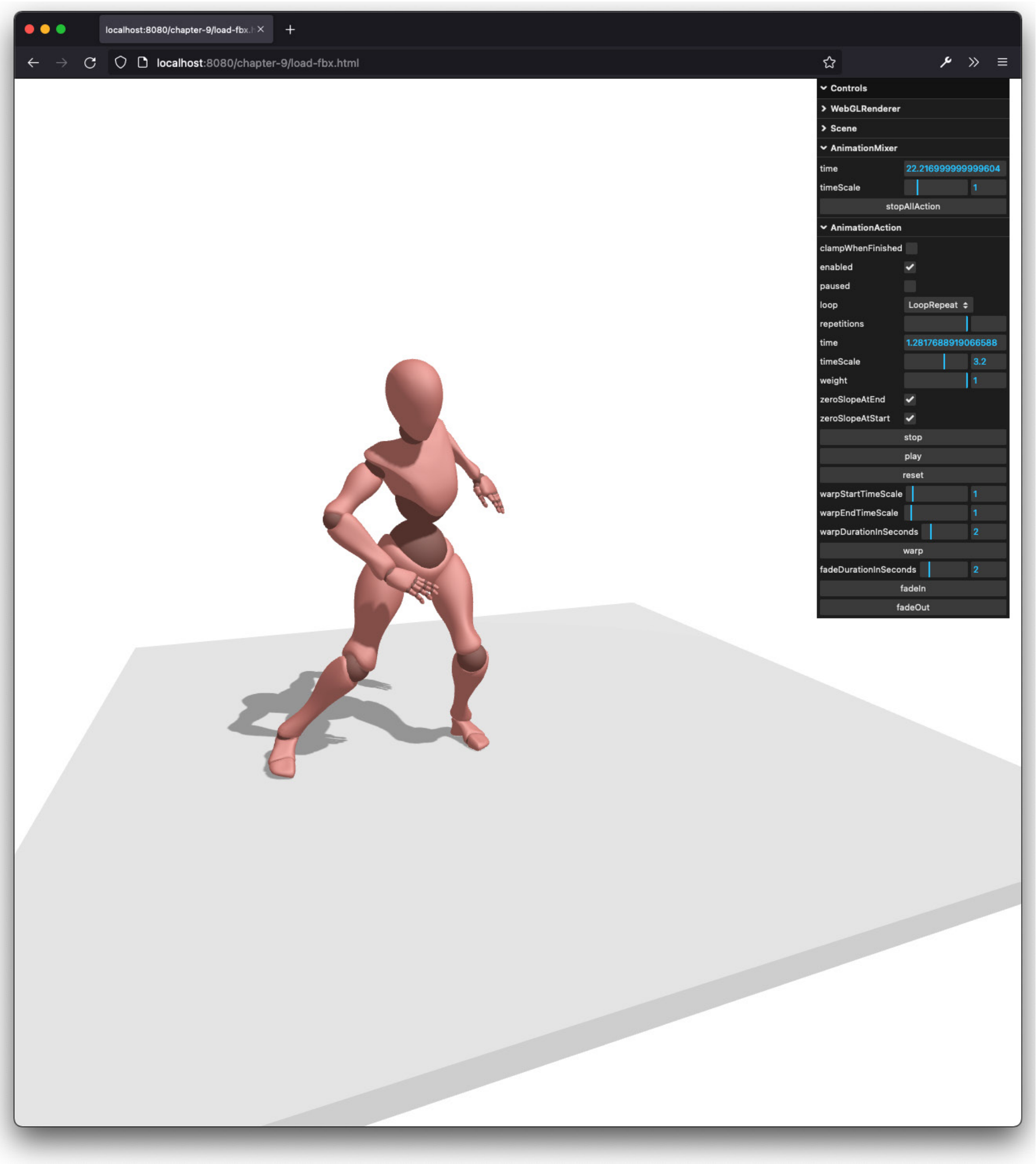

Mixamo
https://www.mixamo.com/#/?genres=Dance&page=1&type=Motion%2CMotionPack
mixamo
Characters
Animations
Jos
Search
48 Per page
FLAIR ON MICHELLE
Bboy Uprock Start
Swing Dancing
Bboy Hip Hop Move
Robot Hip Hop Dance
House Dancing
Flair
Brooklyn Uprock
Shuffling
Hip Hop Dancing
Samba Dancing
Dancing Twerk
Twist Dance
Breakdance Uprock Var 2
Breakdance 1990
Female Dance Pose
DOWNLOAD
SEND TO AERO
UPLOAD CHARACTER
Flair
Overdrive 50
Character Arm-Space 50
Trim 31 total frames
0
100
☐ Mirror
9 / 30

Figure 9.22 – Loading animations from mixamo

After downloading an animation, using it from Three.js is easy:

```
import { FBXLoader } from 'three/examples/jsm/loaders/
FBXLoader'
...
loader.loadAsync('/assets/models/salsa/salsa.fbx').then((mesh)
=> {
  mesh.translateX(-0.8)
  mesh.translateY(-1.9)
  mesh.scale.set(0.03, 0.03, 0.03)
  scene.add(mesh)
  const mixer = new THREE.AnimationMixer(mesh)
  const clips = mesh.animations
  const clip = THREE.AnimationClip.findByName(clips,
    'mixamo.com')
})
```

The resulting animation, as you can see in `load-fbx.html`, looks great:



localhost:8080/chapter-9/load-fbx.
localhost:8080/chapter-9/load-fbx.html
Controls
WebGLRenderer
Scene
AnimationMixer
stopAllAction
AnimationAction
stop
play
reset
warp
fadeIn
fadeOut

time 22.216999999999604
timeScale 1
clampWhenFinished ☐
enabled ☑
paused ☐
loop LoopRepeat
repetitions
time 1.2817688919066588
timeScale 3.2
weight 1
zeroSlopeAtEnd ☑
zeroSlopeAtStart ☑
warpStartTimeScale 1
warpEndTimeScale 1
warpDurationInSeconds 2
fadeDurationInSeconds 2

Figure 9.23 – Animation loaded using fbx

FBX and glTF are modern formats that are used a lot and are a good way to exchange models and animations. There are a couple of older formats around as well. An interesting one is a format used by the old FPS Quake: MD2.

Loading an animation from a Quake model

The MD2 format was created to model characters from Quake, a great game from 1996. Even though the newer engines use a different format, you can still find a lot of interesting models in the MD2 format. Using an MD2 file is a bit different than using the others we've seen so far. When you load an MD2 model, you get a geometry, so you have to make sure that you create a material as well and assign a skin:

```
let animations = []
const loader = new MD2Loader()
loader.loadAsync('/assets/models/ogre/ogro.md2').then
  ((object) => {
  const mat = new THREE.MeshStandardMaterial({
    color: 0xffffff,
    metalness: 0,
    map: new THREE.TextureLoader().load
      ('/assets/models/ogre/skins/skin.jpg')
  })
  animations = object.animations
  const mesh = new THREE.Mesh(object, mat)
  // add to scene, and you can animate it as we've seen
    already
})
```

Once you have this `Mesh`, setting up the animation works in the same way. The result of this animation can be seen here (`load-md2.html`):

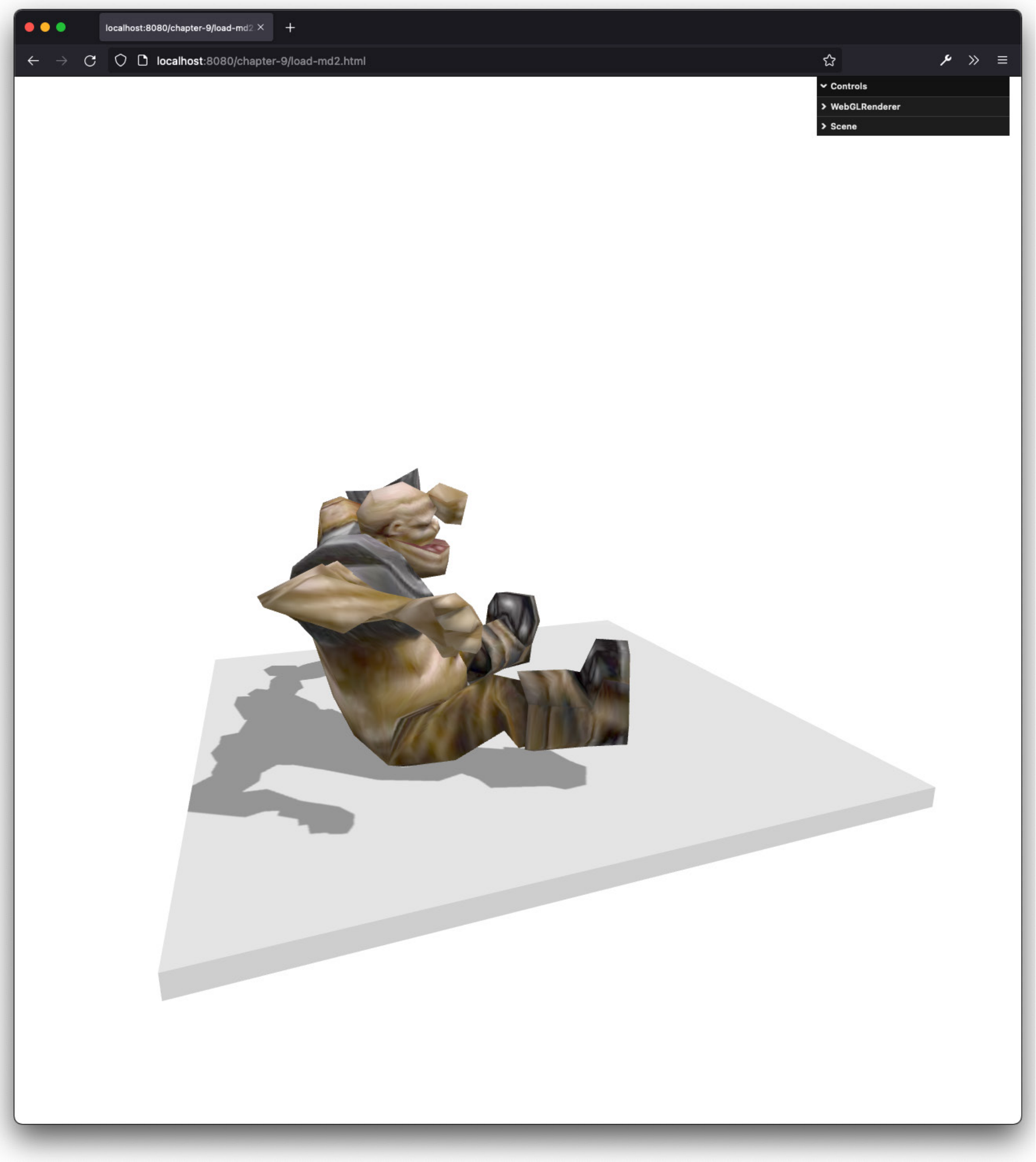

localhost:8080/chapter-9/load-md2
localhost:8080/chapter-9/load-md2.html
Controls
WebGLRenderer
Scene

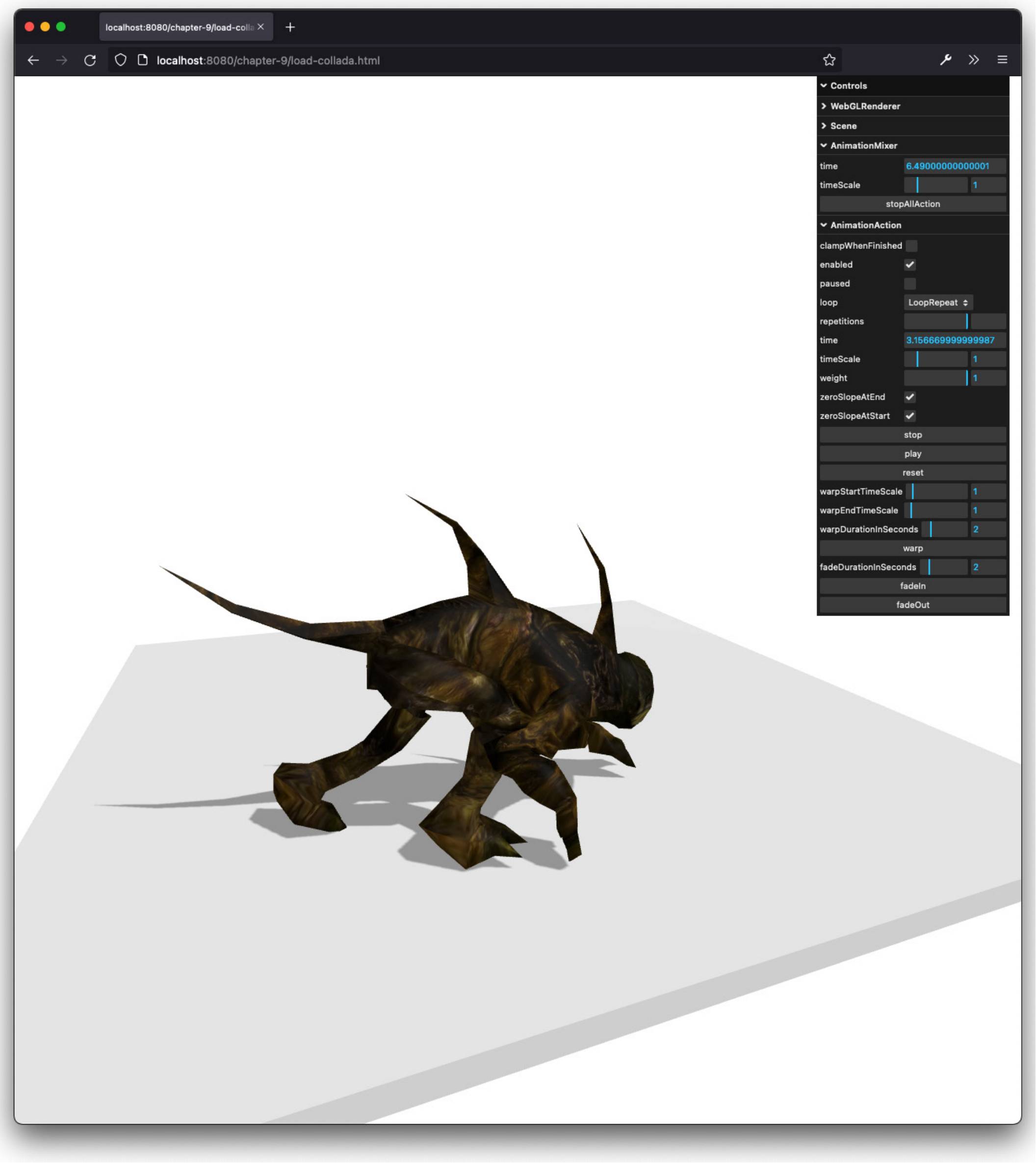
Figure 9.24 – Loaded Quake monster

Next up is COLLADA.

Loading an animation from a COLLADA model

While the normal COLLADA models aren't compressed (and they can get quite large), there is also a `KMZLoader` available in Three.js. This is a compressed COLLADA model, so if you run into

Keyhole Markup Language Zipped (**KMZ**) models, you can load the model using `KMZLoader` instead of `ColladaLoader`:


localhost:8080/chapter-9/load-colla
localhost:8080/chapter-9/load-collada.html
Controls
WebGLRenderer
Scene
AnimationMixer
stopAllAction
AnimationAction
stop
play
reset
warp
fadeIn
fadeOut

time 6.49000000000001
timeScale 1
clampWhenFinished ☐
enabled ☑
paused ☐
loop LoopRepeat
repetitions
time 3.156669999999987
timeScale 1
weight 1
zeroSlopeAtEnd ☑
zeroSlopeAtStart ☑
warpStartTimeScale 1
warpEndTimeScale 1
warpDurationInSeconds 2
fadeDurationInSeconds 2

Figure 9.25 – Loaded COLLADA model

For the final loader, we'll look at `BVHLoader`.

Visualizing a skeleton with BVHLoader

`BVHLoader` is a slightly different loader than the ones we've seen so far. This loader doesn't return meshes or geometries with animations; instead, it returns a skeleton and an animation. An example of this is shown in `load-bvh.html`:

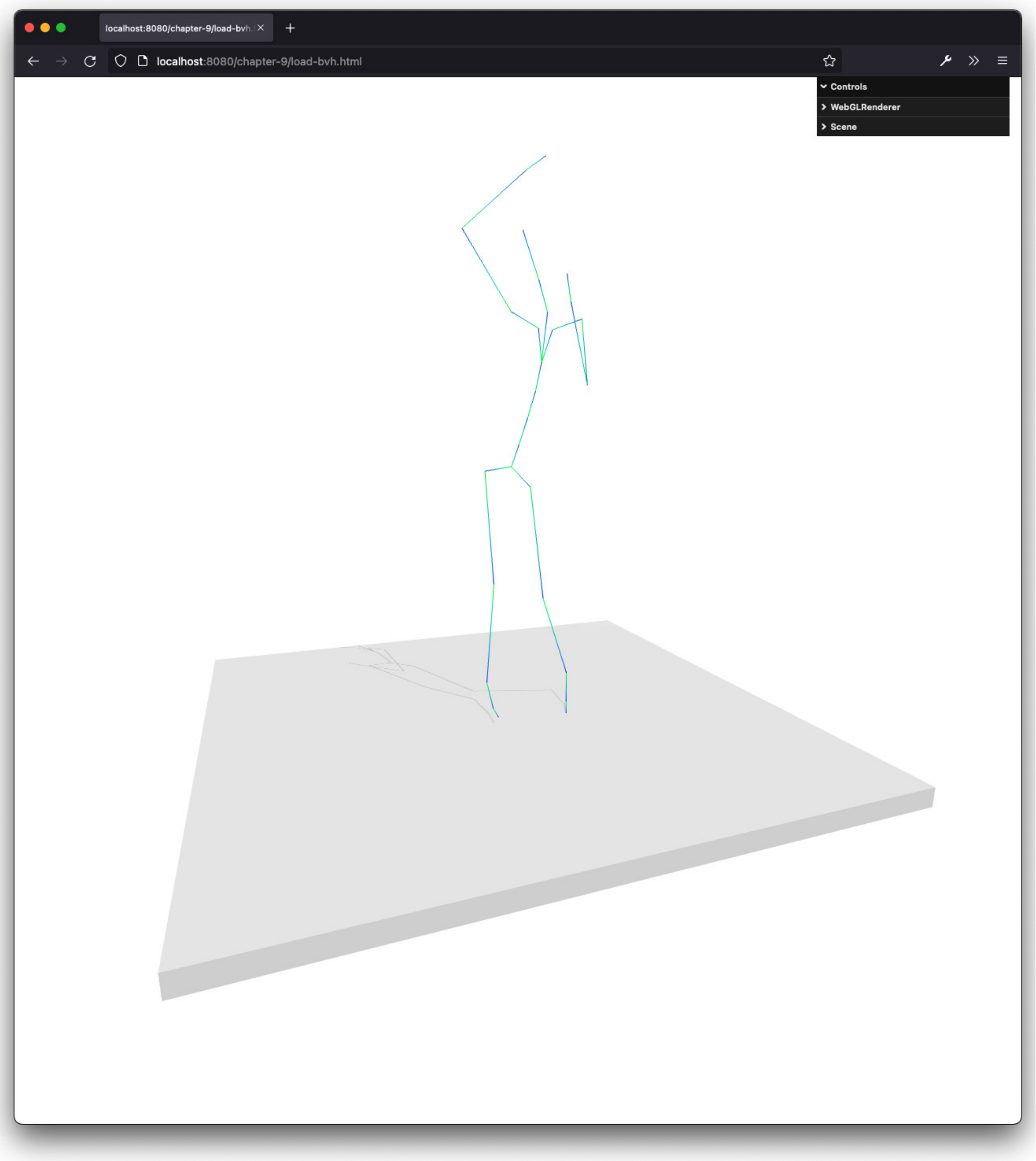

localhost:8080/chapter-9/load-bvh.
localhost:8080/chapter-9/load-bvh.html
Controls
WebGLRenderer
Scene

Figure 9.26 – Loaded BVH skeleton

To visualize this, we can use a `THREE.SkeletonHelper`, as seen here. With a `THREE.SkeletonHelper`, we can visualize the skeleton of a mesh. BVH models just contain skeleton information, which we can visualize like this:

```
const loader = new BVHLoader()
let animation = undefined
loader.loadAsync('/assets/models//amelia-dance/DanceNightClub7_
t1.bvh').then((result) => {
  const skeletonHelper = new THREE.SkeletonHelper
    (result.skeleton.bones[0])
  skeletonHelper.skeleton = result.skeleton
  const boneContainer = new THREE.Group()
  boneContainer.add(result.skeleton.bones[0])
  animation = result.clip
  const group = new THREE.Group()
  group.add(skeletonHelper)
  group.add(boneContainer)
  group.scale.setScalar(0.2)
  group.translateY(-1.6)
  group.translateX(-3)
  // Now we can animate the group just like we did for the
    other examples
})
```

In older versions of Three.js, there was support for other kinds of animation file formats. Most of those are obsolete and have subsequently been removed from the Three.js distribution. If you do stumble upon a different format in which you want to show the animations, you can look at the older Three.js releases and possibly reuse the loaders from there.

Summary

In this chapter, we looked at different ways you can animate your scene. We started with some basic animation tricks, moved on to camera movement and control, and ended by looking at animating models using morph targets and skeleton/bones animations.

When you have the render loop in place, adding simple animations is very easy. Just change a property of the mesh; in the next rendering step, Three.js will render the updated mesh. For more complex animations, you would usually model them in external programs and load them through one of the loaders provided by Three.js.

In the previous chapters, we looked at the various materials we can use to skin our objects. For instance, we saw how we can change the color, shininess, and opacity of these materials. What we haven't discussed in detail yet, however, is how we can use external images (also called textures) together with these materials. With textures, we can easily create objects that look as if they are made out of wood, metal, stone, and much more. In *Chapter 10*, we'll explore all the different aspects of textures and how they are used in Three.js.

10

Loading and Working with Textures

In *Chapter 4*, *Working with Three.js Materials*, we introduced you to the various materials that are available in Three.js. However, we didn't discuss applying textures to the material that's used when creating a mesh. In this chapter, we'll look at that subject. Specifically, we'll discuss the following topics:

- Loading textures in Three.js and applying them to a mesh
- Using bump, normal, and displacement maps to apply depth and detail to a mesh
- Creating fake shadows using a lightmap and an ambient occlusion map
- Using specular, metalness, and roughness maps to set the shininess of specific parts of a mesh
- Applying an alpha map for partial transparency of an object
- Adding detailed reflections to a material using an environment map
- Using the HTML5 Canvas and video element as input for a texture

Let's start with a basic example, where we will show you how to load and apply a texture.

Using textures in materials

There are different ways that textures can be used in Three.js. You can use them to define the colors of the mesh, but you can also use them to define shininess, bumps, and reflections. The first example we will look at, though, is very basic, wherein we will use a texture to define the colors of the individual pixels of a mesh. This is often called a color map or a diffuse map.

Loading a texture and applying it to a mesh

The most basic usage of a texture is when it's set as a map on a material. When you use this material to create a mesh, the mesh will be colored based on the supplied texture. Loading a texture and using it on a mesh can be done in the following manner:

```
const textureLoader = new THREE.TextureLoader();
const texture = textureLoader.load
  ('/assets/textures/ground/ground_0036_color_1k.jpg')
```

In this code sample, we are using an instance of `THREE.TextureLoader` to load an image file from a specific location. Using this loader, you can use PNG, GIF, or JPEG images as input for a texture (later in this chapter, we'll show you how to load other texture formats). Note that textures are loaded asynchronously: if it is a large texture and you render the scene before the texture is completely loaded, you'll see your meshes without a texture applied for a short time. If you want to wait until a texture has been loaded, you can provide a callback to the `textureLoader.load()` function:

```
Const textureLoader = new THREE.TextureLoader();
const texture = textureLoader.load
  ('/assets/textures/ground/ground_0036_color_1k.jpg',
            onLoadFunction,
            onProgressFunction,
            onErrorFunction)
```

As you can see, the `load` function takes three additional functions as parameters: `onLoadFunction` is called when the texture is loaded, `onProgressFunction` can be used to track how much of the texture is loaded, and `onErrorFunction` is called when something goes wrong while loading or parsing the texture. Now that the texture has been loaded, we can add it to the mesh:

```
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.map = texture
```

Note that the loader also provides a `loadAsync` function, which returns a `Promise` instead, just like we saw in the previous chapter when loading models.

You can use pretty much any image you'd like as a texture. However, you'll get the best results by using a square texture whose dimensions are a power of 2. So, dimensions such as 256 x 256, 512 x 512, 1,024 x 1,024, and so on work the best. If the texture isn't a power of two, Three.js will scale down the image to the closest power of 2 value.

One of the textures we'll be using in the examples in this chapter looks like this:

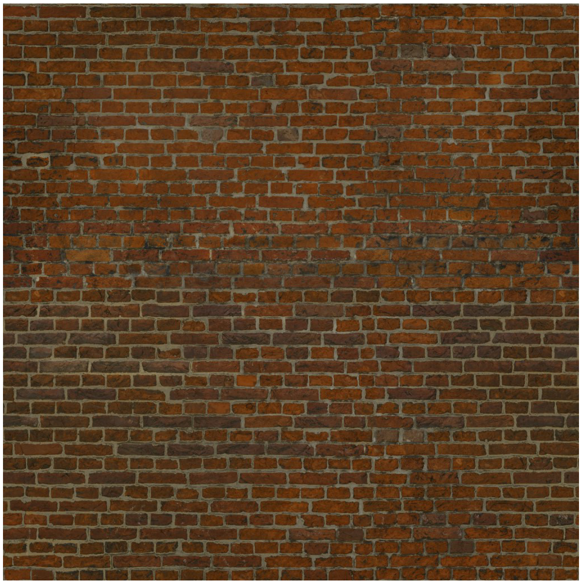


Figure 10.1 – The color texture of a brick wall

The pixels of a texture (also called texels) usually don't map one-to-one on the pixels of the face. If the camera is very close, we need to magnify the texture, and if we're zoomed out, we probably need to shrink the texture. For this purpose, WebGL and Three.js offer a couple of different options to resize this image. This is done through the `magFilter` and `minFilter` properties:

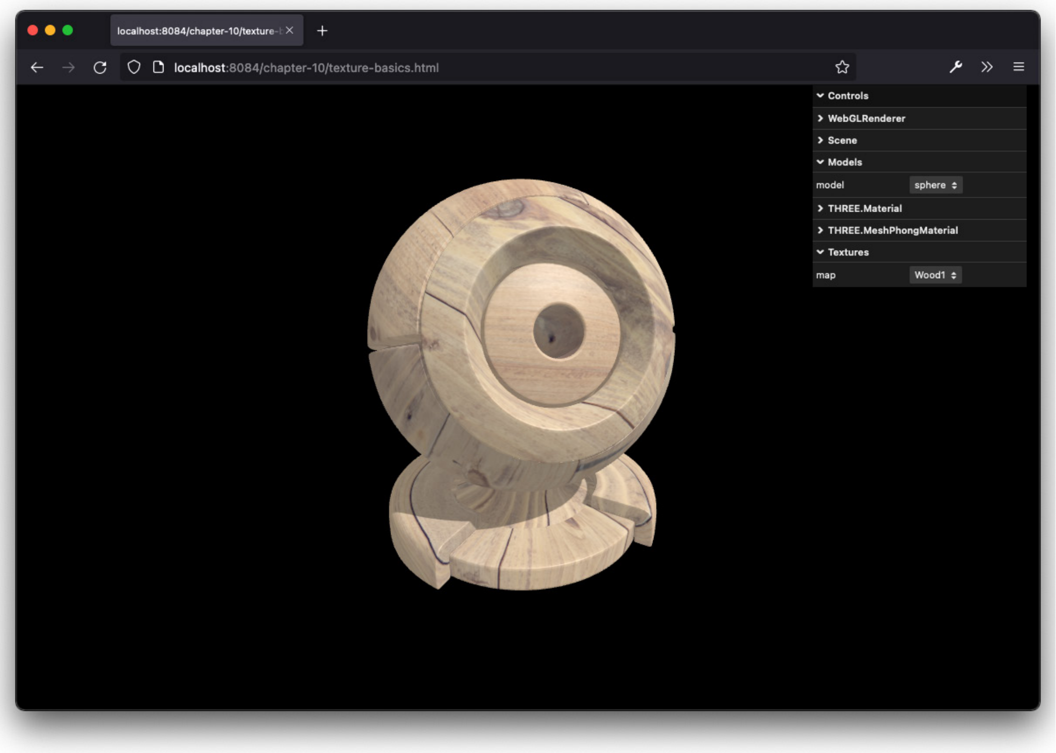
- `THREE.NearestFilter`: This filter uses the color of the nearest texel that it can find. When used for magnification, this will result in blockiness, and when used for minification, the result will lose a lot of detail.
- `THREE.LinearFilter`: This filter is more advanced; it uses the color values of the four neighboring texels to determine the correct color. You'll still lose a lot of detail in minification, but the magnification will be much smoother and less blocky.

Aside from these basic values, we can also use a **MIP map**. A MIP map is a set of texture images, each half the size of the previous one. These are created when you load the texture and allow for much smoother filtering. So, when you have a square texture (as a power of 2), you can use a couple of additional approaches for better filtering. The properties can be set using the following values:

- `THREE.NearestMipMapNearestFilter`: This property selects the MIP map that best maps the required resolution and applies the nearest filter principle, which we discussed in the previous list. Magnification is still blocky, but minification looks much better.
- `THREE.NearestMipMapLinearFilter`: This property selects not just a single MIP map but the two nearest MIP map levels. On both of these levels, the nearest filter is applied, to get two intermediate results. These two results are passed through a linear filter to get the final result.
- `THREE.LinearMipMapNearestFilter`: This property selects the MIP map that best maps the required resolution and applies the linear filter principle, which was discussed in the previous list.
- `THREE.LinearMipMapLinearFilter`: This property selects not a single MIP map, but the two nearest MIP map levels. On both of these levels, a linear filter is applied, to get two intermediate results. These two results are passed through a linear filter to get the final result.

If you don't specify the `magFilter` and `minFilter` properties explicitly, Three.js uses `THREE.LinearFilter` as the default for the `magFilter` property and `THREE.LinearMipMapLinearFilter` as the default for the `minFilter` property.

In our examples, we'll just use the default texture properties. An example of such a basic texture used as a map for a material can be found in `texture-basics.html`. The following screenshot shows this example:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-basics.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

model sphere

map Wood1

Figure 10.2 – Model with a simple wood texture

In this example, you can change the models and select a couple of textures from the menu on the right. You can also change the default material properties to see how the material, in combination with a color map, is affected by the different settings.

In this example, you can see that the textures wrap nicely around the shapes. When you create geometries in Three.js, it makes sure that any texture that is used is applied correctly. This is done through something called UV mapping. With UV mapping, we can tell the renderer which part of a texture should be applied to a specific face. We'll get into the details of UV mapping in *Chapter 13*, *Working with Blender and Three.js*, where we will show you how you can easily use Blender to create custom UV mappings for Three.js.

Aside from the standard image formats that we can load with `THREE.TextureLoader`, Three.js also provides a couple of custom loaders you can use to load textures provided in different formats. If you've got a specific image format, you can check out the `loaders` folder from the Three.js distribution (`https://github.com/mrdoob/three.js/tree/dev/examples/jsm/loaders`) to see whether the image format can be loaded directly by Three.js or whether you need to manually convert it.

Aside from these normal images, Three.js also supports HDR images.

Loading HDR images as textures

An HDR image captures a higher range of luminance levels than standard images, and can more closely match what we see with the human eye. Three.js supports the `EXR` and `RGBE` formats. If you've got an HDR image, you can fine-tune how Three.js renders the HDR image since an HDR image contains more luminance information than can be shown on a display. This can be done by setting the following properties in `THREE.WebGLRenderer`:

- `toneMapping`: This property defines how to map the colors from the HDR image to the display. Three.js provides the following options: `THREE.NoToneMapping`, `THREE.LinearToneMapping`, `THREE.ReinhardToneMapping`, `THREE.Uncharted2ToneMapping`, and `THREE.CineonToneMapping`. The default is `THREE.LinearToneMapping`.
- `toneMappingExposure`: This is the exposure level of `toneMapping`. This can be used to fine-tune the colors of the rendered texture.
- `toneMappingWhitePoint`: This is the white point used for `toneMapping`. This can also be used to fine-tune the colors of the rendered texture.

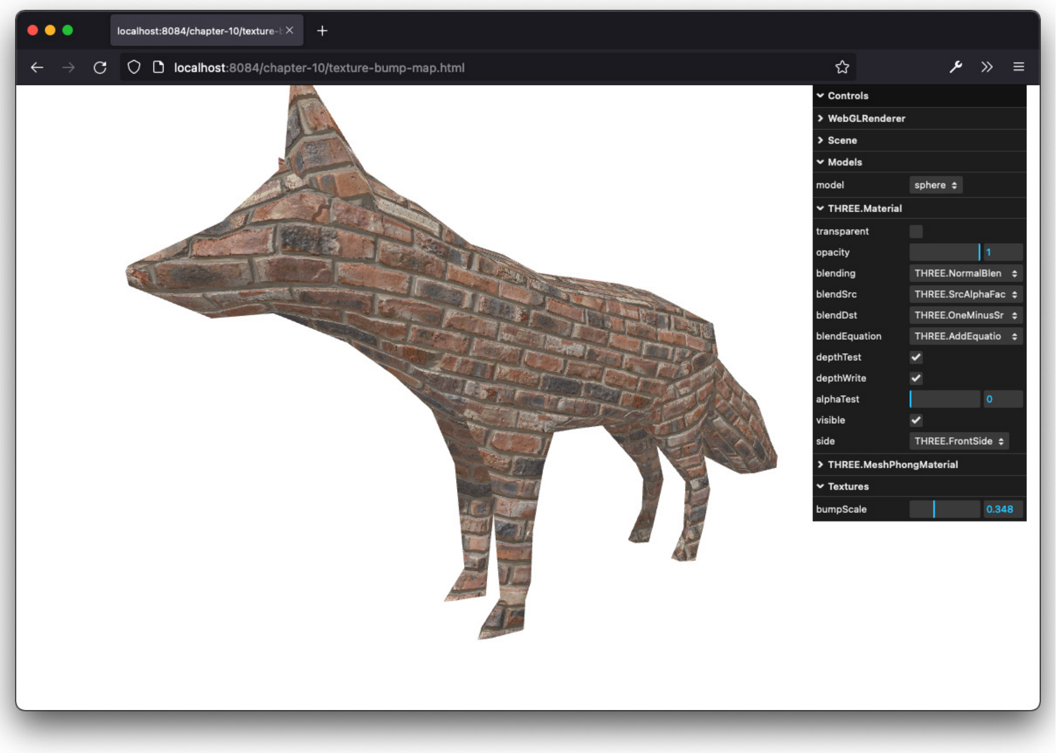
If you want to load an EXR or RGBE image and use it as a texture, you can use `THREE.EXRLoader` or `THREE.RGBELoader`. This works in the same way as we've seen for `THREE.TextureLoader`:

```
const loader = new THREE.EXRLoader();
exrTextureLoader.load('/assets/textures/exr/Rec709.exr')
...
const hdrTextureLoader = new THREE.RGBELoader();
hdrTextureLoader.load('/assets/textures/hdr/
  dani_cathedral_oBBC.hdr')
```

In the `texture-basics.html` example, we showed you how the texture can be used to apply colors to a mesh. In the next section, we'll look at how to use a texture to make a model look more detailed by applying fake height information to the mesh.

Using a bump map to provide extra details to a mesh

A bump map is used to add more depth to a material. You can see this in action by opening `texture-bump-map.html`:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-bump-map.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
bumpScale 0.348

Figure 10.3 – Model with a bump map

In this example, you can see that the model looks much more detailed and seems to have more depth. This was done by setting an additional texture, a so-called bump map, on the material:

```
const exrLoader = new EXRLoader()
const colorMap = exrLoader.load('/assets/textures/brick-wall/
brick_wall_001_diffuse_2k.exr', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
const bumpMap = new THREE.TextureLoader().load(
  '/assets/textures/brick-wall/brick_wall_001_displacement_2k.
png',
  (texture) => {
    texture.wrapS = THREE.RepeatWrapping
    texture.wrapT = THREE.RepeatWrapping
    texture.repeat.set(4, 4)
  }
)
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.map = colorMap
material.bumpMap = bumpMap
```

In this code, you can see that aside from setting the map property, we have also set the `bumpMap` property to a texture. Additionally, with the `bumpScale` property, which is available through the menu in the previous example, we can set the height (or depth, if set to a negative value) of the bumps. The textures used in this example are shown here:

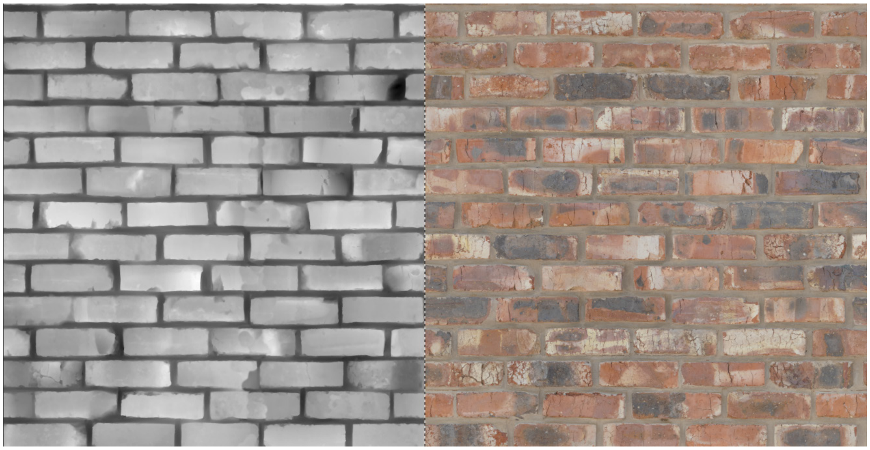
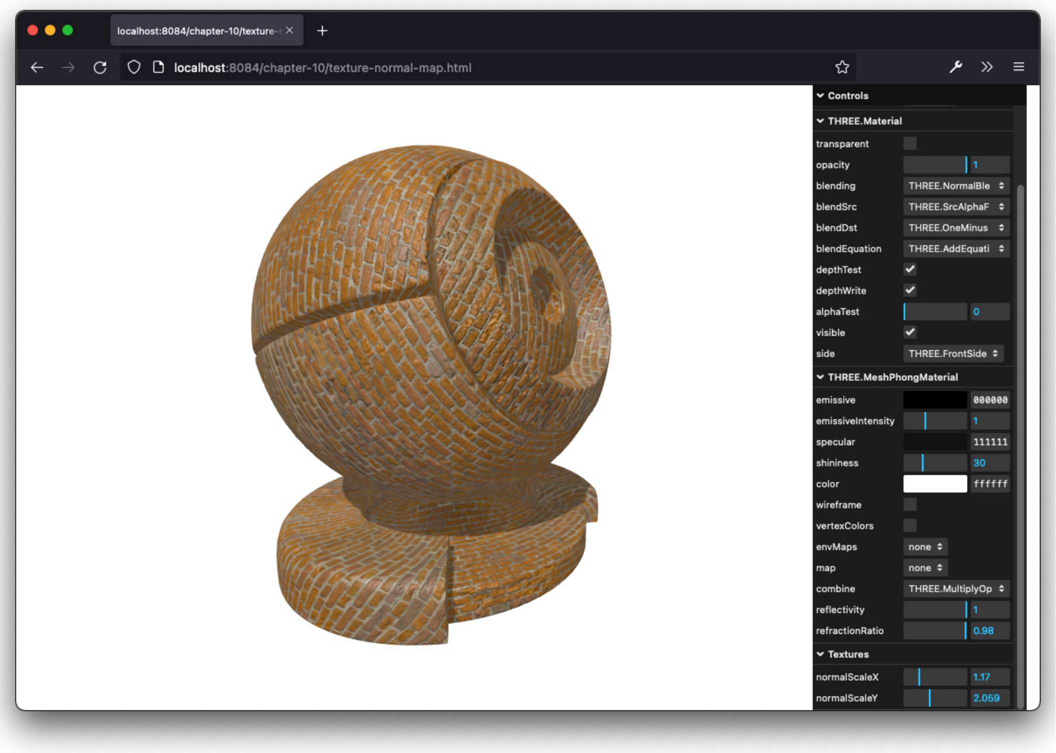


Figure 10.4 – Textures used for the bump map

The bump map is a grayscale image, but you can also use a color image. The intensity of the pixels defines the height of the bump. A bump map only contains the relative height of a pixel. It doesn't say anything about the direction of the slope. So, the level of detail and perception of depth that you can reach with a bump map is limited. For more detail, you can use a normal map.

Achieving more detailed bumps and wrinkles with a normal map

In a normal map, the height (displacement) is not stored, but the direction of the normal for each pixel is stored. Without going into too much detail, with normal maps, you can create very detailed-looking models that use only a small number of vertices and faces. For instance, take a look at the `texture-normal-map.html` example:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-normal-map.html

Controls

THREE.Material

transparent	☐
opacity	1
blending	THREE.NormalBle
blendSrc	THREE.SrcAlphaF
blendDst	THREE.OneMinus
blendEquation	THREE.AddEquati
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide

THREE.MeshPhongMaterial

emissive	000000
emissiveIntensity	1
specular	111111
shininess	30
color	ffffff
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	THREE.MultiplyOp
reflectivity	1
refractionRatio	0.98

Textures

normalScaleX	1.17
normalScaleY	2.059

Figure 10.5 – A model using a normal map

In the preceding screenshot, you can see a very detailed-looking model. And as the model moves around, you can see that the texture is responding to the light it receives. This provides a very realistic-looking model and only requires a very simple model and a couple of textures. The following code fragment shows how to use a normal map in Three.js:

```
const colorMap = new THREE.TextureLoader().load('/assets/
textures/red-bricks/red_bricks_04_diff_1k.jpg', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
const normalMap = new THREE.TextureLoader().load(
  '/assets/textures/red-bricks/red_bricks_04_nor_gl_1k.jpg',
  (texture) => {
    texture.wrapS = THREE.RepeatWrapping
    texture.wrapT = THREE.RepeatWrapping
    texture.repeat.set(4, 4)
  }
)
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.map = colorMap
material.normalMap = normalMap
```

This involves the same approach as the one we used for the bump map. This time, though, we set the `normalMap` property to the normal texture. We can also define how pronounced the bumps look by setting the `normalScale` property (`mat.normalScale.set(1,1)`). With this property, you can scale along the *X* and *Y* axes. The best approach, though, is to keep these values the same. In this example, you can play around with these values.

The following figure shows what the normal map we used here looks like:

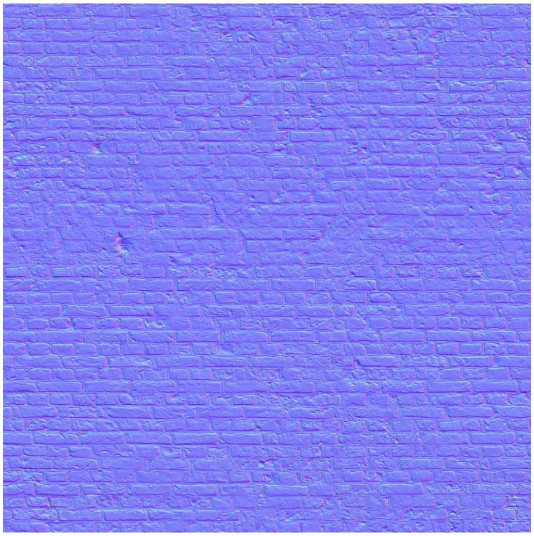


Figure 10.6 – Normal texture

The problem with normal maps, however, is that they aren't very easy to create. You need to use specialized tools, such as Blender or Photoshop. These programs can use high-resolution renderings or textures as input and can create normal maps from them.

With a normal or bump map, you don't change the shape of the model; all of the vertices stay in the same location. These maps just use the lights from the scene to create fake depth and details. However, Three.js provides a third method that you can use to add details to a model using a map, which does change the positions of the vertices. This is done through a displacement map.

Using a displacement map to alter the position of vertices

Three.js also provides a texture that you can use to change the positions of the vertices of your model. While the bump map and the normal map give an illusion of depth, with a displacement map, we change the model's shape, based on the information from the texture. We can use a displacement map in the same way as we use other maps:

```
const colorMap = new THREE.TextureLoader().load('/assets/
textures/displacement
  /w_c.jpg', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
})
const displacementMap = new THREE.TextureLoader().load('/
assets/textures/displacement
  /w_d.png', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
})
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.map = colorMap
material.displacementMap = displacementMap
```

In the preceding code fragment, we loaded a displacement map, which looks as follows:

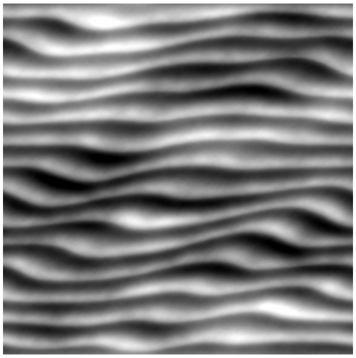
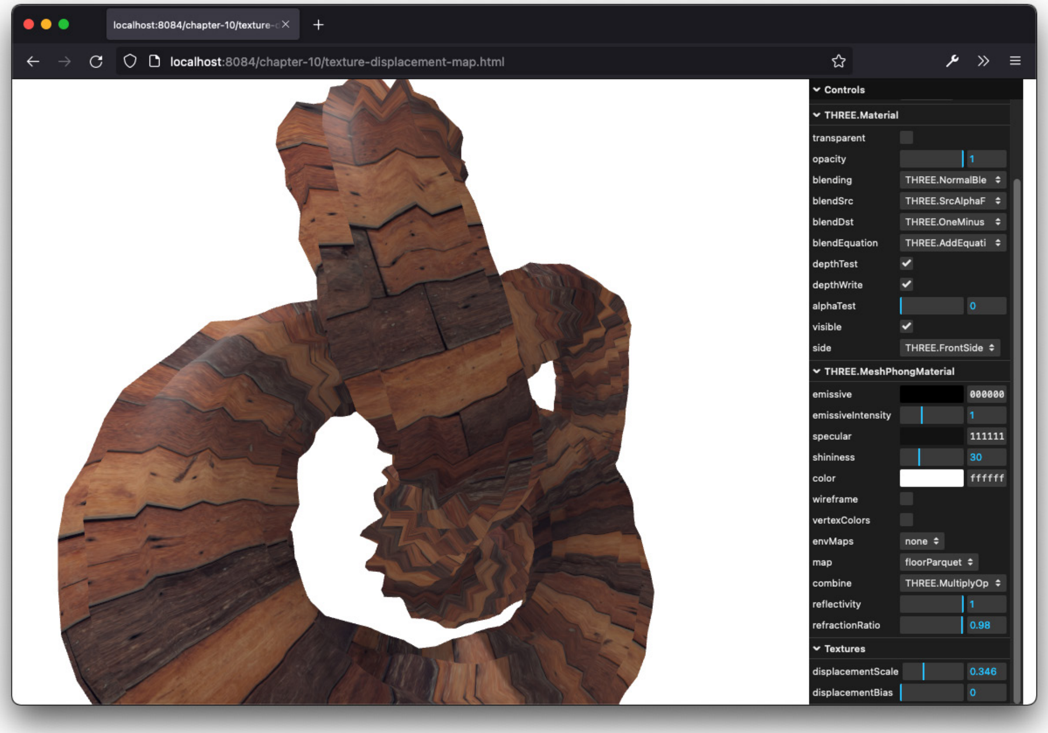


Figure 10.7 – Displacement map

The brighter the color, the more a vertex is displaced. When you run the `texture-displacement.html` example, you will see that the result of the displacement map is a model where the shape of the model is changed based on the information from the map:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-displacement-map.html

Controls

THREE.Material

transparent	☐
opacity	1
blending	THREE.NormalBle
blendSrc	THREE.SrcAlphaF
blendDst	THREE.OneMinus
blendEquation	THREE.AddEquati
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide

THREE.MeshPhongMaterial

emissive	000000
emissiveIntensity	1
specular	111111
shininess	30
color	ffffff
wireframe	☐
vertexColors	☐
envMaps	none
map	floorParquet
combine	THREE.MultiplyOp
reflectivity	1
refractionRatio	0.98

Textures

displacementScale	0.346
displacementBias	0

Figure 10.8 – Model using a displacement map

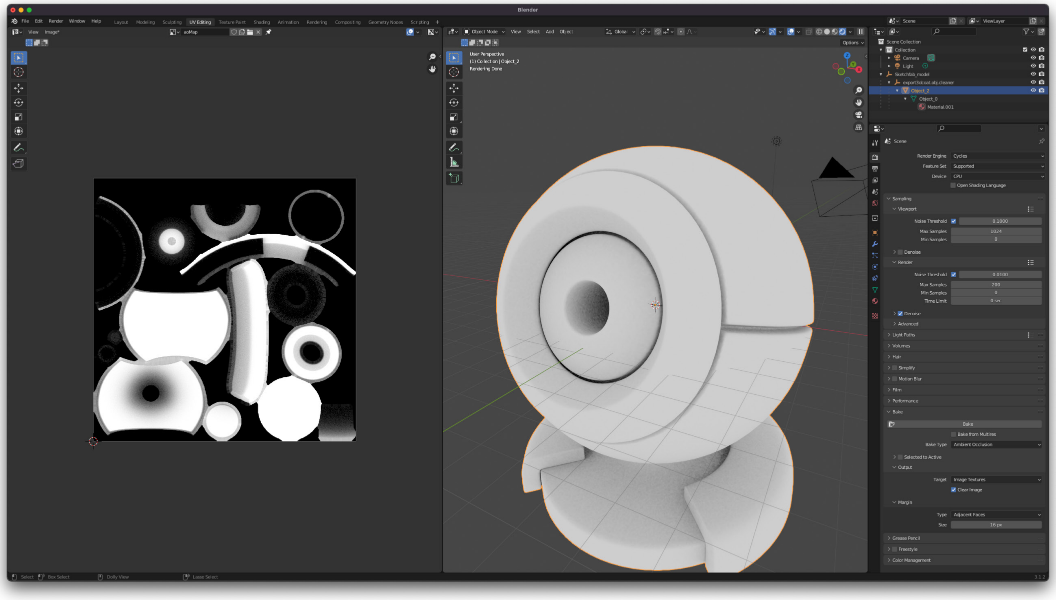
Aside from setting the `displacementMap` texture, we can also use `displacementScale` and `displacementOffset` to control how pronounced the displacement is. One final thing to mention about using a displacement map is that it will only have good results if your mesh contains a large number of vertices. If not, the displacement won't look like the provided texture, since there are too few vertices to represent the required displacement.

Adding subtle shadows with an ambient occlusion map

In the previous chapters, you learned how you can use shadows in Three.js. If you set the `castShadow` and `receiveShadow` properties of the correct meshes, add a couple of lights, and configure the shadow camera of the lights correctly, Three.js will render shadows.

Rendering shadows, however, is a rather expensive operation that is repeated for every render loop. If you have lights or objects that are moving around, this is necessary, but often, some of the lights or models are fixed, so it would be great if we could calculate the shadows once, and then reuse them. To accomplish this, Three.js offers two different maps: the ambient occlusion map and a lightmap. In this section, we'll look at the ambient occlusion map, and in the next section, we'll look at the lightmap.

Ambient occlusion is a technique used to determine how much each part of a model is exposed to the ambient lighting in a scene. In tools such as Blender, ambient light is often modeled through a hemisphere light or a directional light, such as the Sun. While most parts of a model will receive some of this ambient lighting, not all of the parts will receive the same. If, for instance, you model a person, the top of the head will receive more ambient lighting than the bottom of the arms. This difference in lighting – the shadows – can be rendered (baked, as shown in the following screenshot) into a texture, and we can then apply that texture to our models to give them shadows, without having to calculate the shadows every time:


Blender
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes Scripting
View Image*
aoMap
Object Mode View Select Add Object
Global
Options
User Perspective
(1) Collection | Object_2
Rendering Done
Scene
ViewLayer
Scene Collection
Collection
Camera
Light
Sketchfab_model
export3dcoat.obj.cleaner
Object_2
Object_0
Material.001
Sampling
Viewport
Render
Advanced
Light Paths
Volumes
Hair
Simplify
Motion Blur
Film
Performance
Bake
Output
Margin
Grease Pencil
Freestyle
Color Management
Select
Box Select
Dolly View
Lasso Select
3.1.2

Render Engine: Cycles
Feature Set: Supported
Device: CPU
☐ Open Shading Language

Viewport: Noise Threshold ☑ 0.1000, Max Samples 1024, Min Samples 0, ☐ Denoise

Render: Noise Threshold ☑ 0.0100, Max Samples 200, Min Samples 0, Time Limit 0 sec, ☑ Denoise

Bake
☐ Bake from Multires
Bake Type: Ambient Occlusion
☐ Selected to Active
Target: Image Textures
☑ Clear Image
Margin Type: Adjacent Faces
Size: 16 px

Figure 10.9 – Ambient occlusion map baked in Blender

Once you have an ambient occlusion map, you can assign it to the `aoMap` property of the material, and Three.js will take this information into account when applying and calculating how much the lights in the scene should be applied to that specific part of the model. The following code fragment shows how to set the `aoMap` property:

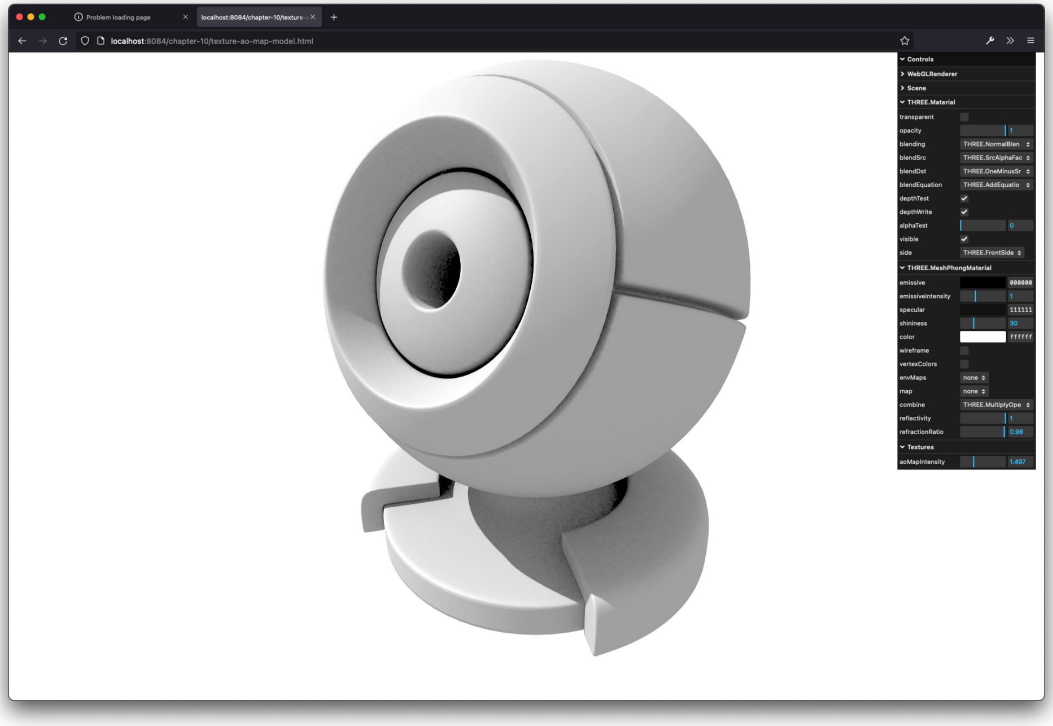
```
const aoMap = new THREE.TextureLoader().load('/assets/gltf/
material_
  ball_in_3d-coat/aoMap.png')
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.aoMap = aoMap
material.aoMap.flipY = false
```

Just like the other kinds of texture maps, we just use `THREE.TextureLoader` to load the texture and assign it to the correct property of the material. And like with many of the other maps, we can also tune how much the map affects the lighting of the model by setting the `aoMapIntenisty` property. In this example, you can also see that we needed to set the `flipY` property of `aoMap` to false. Sometimes, external programs store the material in a texture slightly different than Three.js expects. With this property, we flip the orientation of the texture. This is usually something you'll notice by trial and error when working with the model.

To make an ambient occlusion map work, we will (usually) need one additional step. We have already mentioned UV mappings (stored in the `uv` attribute). These define which part of a texture is mapped to a specific face of the model. For the ambient occlusion map, and also for the lightmap in the following example, Three.js uses a separate set of UV mappings (stored in the `uv2` attribute) because, often, the other textures need to be applied differently than the shadow and lightmap textures. For our example, we are just copying the UV mappings from the model; remember that when we use the `aoMap` property or the `lightMap` property, Three.js will use the value of the `uv2` attribute, instead of the `uv` attribute. If this attribute isn't present in the model you load, most often, just copying the `uv` map property works as well since we didn't do anything to optimize the ambient occlusion map, which might require a different set of UVs:

```
const k = mesh.geometry
const uv1 = k.getAttribute('uv')
const uv2 = uv1.clone()
k.setAttribute('uv2', uv2)
```

We will provide two examples where we use an ambient occlusion map. In the first one, we are showing the model from *Figure 10.9* with `aoMap` applied (`texture-ao-map-model.html`):


Problem loading page
localhost:8084/chapter-10/texture-a
localhost:8084/chapter-10/texture-ao-map-model.html
Controls
WebGLRenderer
Scene

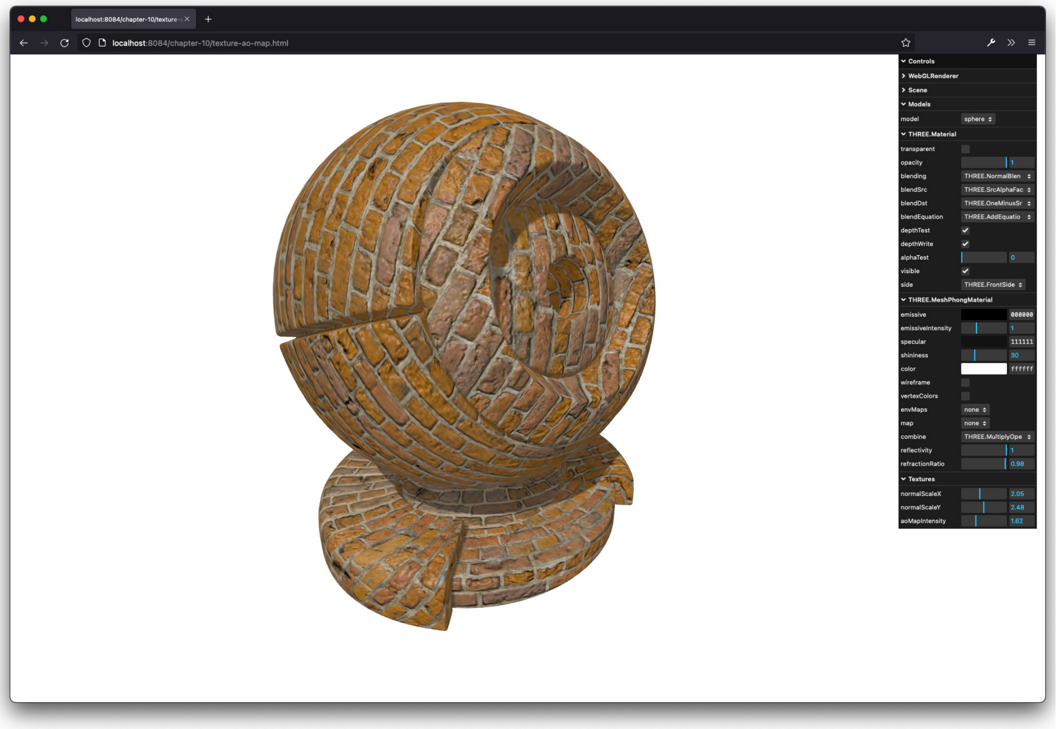
THREE.Material
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide

THREE.MeshPhongMaterial
emissive 000000
emissiveIntensity 1
specular 111111
shininess 30
color ffffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98

Textures
aoMapIntensity 1.497

Figure 10.10 – Ambient occlusion map baked in Blender and then applied to a model

You can use the menu on the right to set `aoMapIntensity`. The higher this value is, the more shadows you'll see from the loaded `aoMap` texture. As you can see, it is really useful to have an ambient occlusion map as it provides great details for the model, and makes it look much more lifelike. Some of the textures we've seen in this chapter already also provide an additional `aoMap` that you can use. If you open `texture-ao-map.html`, you will get a simple brick-like texture, but this time with `aoMap` added as well:


localhost:8084/chapter-10/texture-a
localhost:8084/chapter-10/texture-ao-map.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

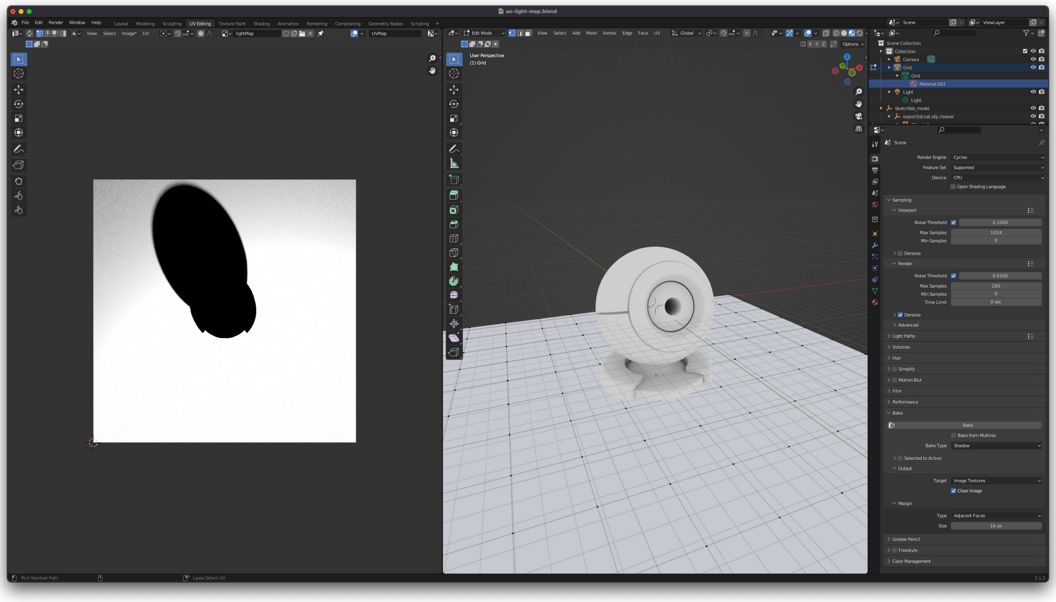
model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 1
specular 111111
shininess 30
color ffffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98
normalScaleX 2.05
normalScaleY 2.48
aoMapIntensity 1.62

Figure 10.11 – Ambient occlusion map combined with color and normal maps

While an ambient occlusion map changes the amount of light received by certain parts of the model, Three.js also supports `lightmap`, which do the opposite (approximately) by specifying a map that adds extra lighting to certain parts of the model.

Creating fake lighting using a lightmap

In this section, we'll use a lightmap. A lightmap is a texture that contains information about how much the lights in the scene will affect the model. In other words, the effect of the lights is baked into a texture. Lightmaps are baked in 3D software, such as Blender, and contain the light values of each part of the model:


ao-light-map.blend
User Perspective
(1) Grid
Collection
Camera
Grid
Grid
Material.002
Light
Light
Sketchfab_model
export3dcoat.obj.cleaner
Scene
Sampling
Viewport
Render
Light Paths
Volumes
Hair
Simplify
Motion Blur
Film
Performance
Bake
Output
Margin
Grease Pencil
Freestyle
Color Management
3.1.2

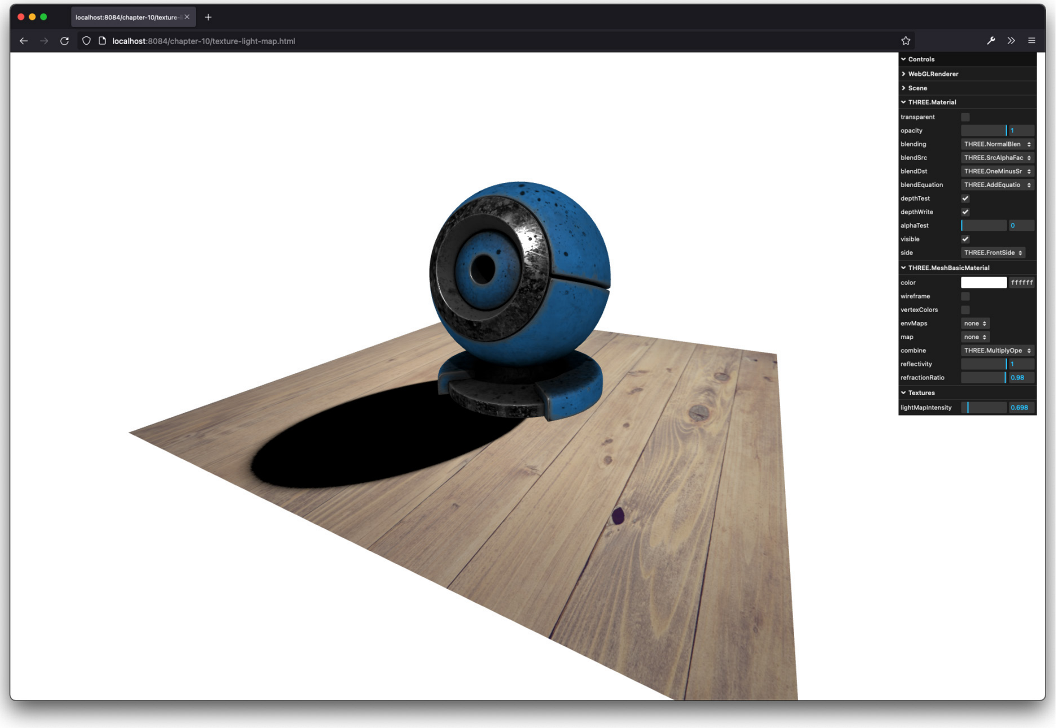
Render Engine: Cycles
Feature Set: Supported
Device: CPU
☐ Open Shading Language
Viewport Noise Threshold: ☑ 0.1000
Max Samples: 1024
Min Samples: 0
☐ Denoise
Render Noise Threshold: ☑ 0.0100
Max Samples: 200
Min Samples: 0
Time Limit: 0 sec
☑ Denoise
☐ Bake from Multires
Bake Type: Shadow
☐ Selected to Active
Target: Image Textures
☑ Clear Image
Type: Adjacent Faces
Size: 16 px

Figure 10.12 – Lightmap baked in Blender

The lightmap that we'll use in this example is shown in *Figure 10.12*. The right part of the edit window shows a baked lightmap for the ground plane. You can see that the whole ground plane is illuminated with white light, and parts of it receive less light because there is also a model in the scene. The code for using a lightmap is similar to that of an ambient occlusion map:

```
Const textureLoader = new THREE.TextureLoader()
const colorMap = textureLoader.load('/assets/textures/wood/
  abstract-antique-backdrop-164005.jpg')
const lightMap = textureLoader.load('/assets/gltf/
  material_ball_in_3d-coat/lightMap.png')
const material = new THREE.MeshBasicMaterial({ color:
  0xffffff })
material.map = colorMap
material.lightMap = lightMap
material.lightMap.flipY = false
```

Once again, we need to provide Three.js with an additional set of uv values called `uv2` (not shown in the code), and we must use `THREE.TextureLoader` to load the textures – in this case, a simple texture is used for the colors of the floor and the lightmap created for this example in Blender. The result looks as follows (`texture-light-map.html`):


localhost:8084/chapter-10/texture-l
localhost:8084/chapter-10/texture-light-map.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshBasicMaterial
Textures

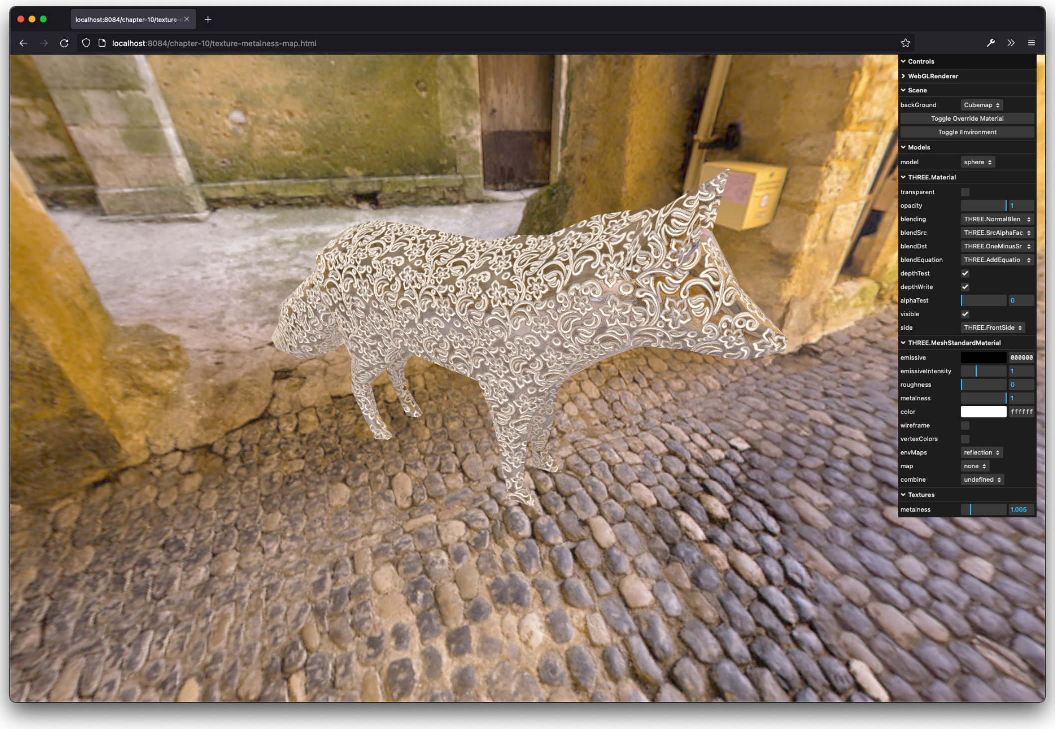
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
color ffffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98
lightMapIntensity 0.698

Figure 10.13 – Using a lightmap for false shadows

If you look at the preceding example, you will see that the information from the lightmap is used to create a very nice-looking shadow, which seems to be cast by the model. It is important to remember that baking shadows, lights, and ambient occlusion works great in static scenes with static objects. As soon as objects or light sources change or start to move, you will have to calculate the shadows in real time.

Metalness and roughness maps

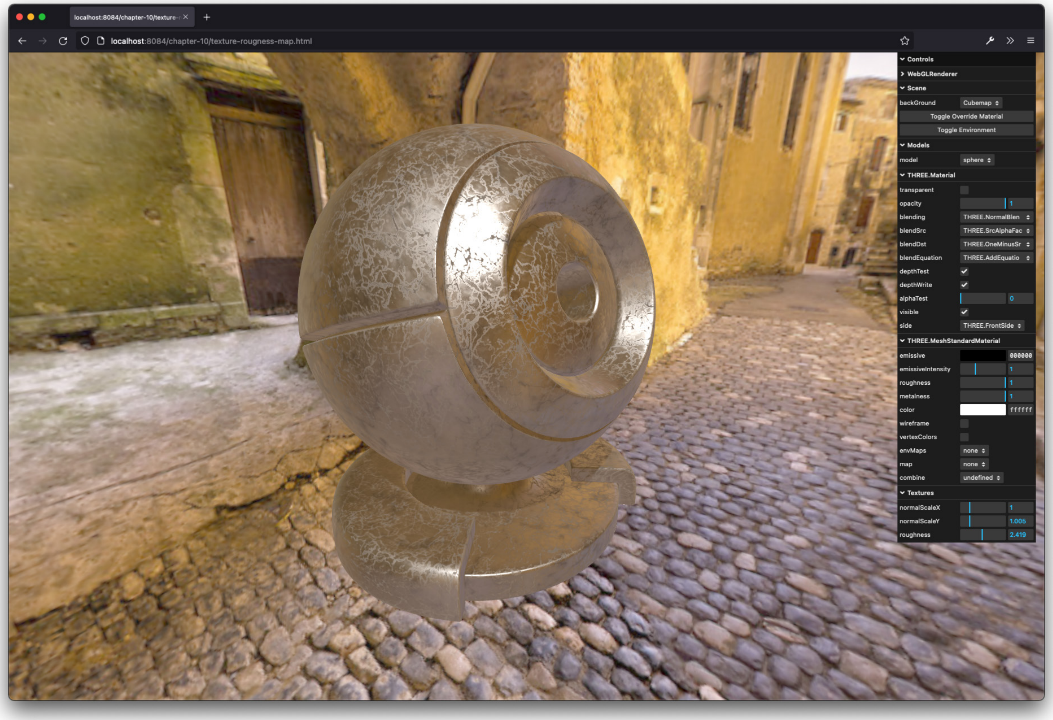
When discussing the materials available in Three.js, we mentioned that a good default material to use is `THREE.MeshStandardMaterial`. You can use this to create shiny, metal-like materials, but also to apply roughness, to make the mesh look more like wood or plastic. By using the metalness and roughness properties of the material, we can configure the material to represent the material that we want. Aside from these two properties, you can also configure these properties by using a texture. So, if we have a rough object and we want to specify that a certain part of that object is shiny, we can set the `metalnessMap` property of `THREE.MeshStandardMaterial`, and if we want to indicate that some parts of the mesh should be seen as scratched or rougher, we can set the `roughnessMap` property. When you use these maps, the value of the texture for a specific part of the model is multiplied by either the `roughness` property or the `metalness` property, and that determines how that specific pixel should be rendered. First, we will look at the `metalness` property in `texture-metalness-map.html`:


localhost:8084/chapter-10/texture-m
localhost:8084/chapter-10/texture-metalness-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshStandardMaterial
Textures

backGround Cubemap
model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ✓
depthWrite ✓
alphaTest 0
visible ✓
side THREE.FrontSide
emissive 000000
emissiveIntensity 1
roughness 0
metalness 1
color ffffff
wireframe ☐
vertexColors ☐
envMaps reflection
map none
combine undefined
metalness 1.005

Figure 10.14 – Metalness texture applied to a model

In this example, we've skipped ahead a bit and have also used an environment map, which allows us to render reflections from the environment on top of the objects. An object with a high metalness reflects more, and an object with a high roughness diffuses the reflection more. For this model, we've used `metalnessMap`; you can see that the object itself is shiny where the `metalness` property from the texture is high and that some parts are rough where the `metalness` property from the texture is low. When looking at `roughnessMap`, we can see pretty much the same but inverted:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-rougness-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshStandardMaterial
Textures

backGround Cubemap
model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 1
roughness 1
metalness 1
color ffffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine undefined
normalScaleX 1
normalScaleY 1.005
roughness 2.419

Figure 10.15 – Roughness texture applied to a model

As you can see, based on the provided texture, certain parts of the model are rougher or more scratched than other parts. For `metalnessMap`, the value of the material is multiplied by the `metalness` property of the material; for `roughnessMap`, the same applies, but in that case, the value is multiplied by the `roughness` property.

Loading these textures and setting them to the material can be done like so:

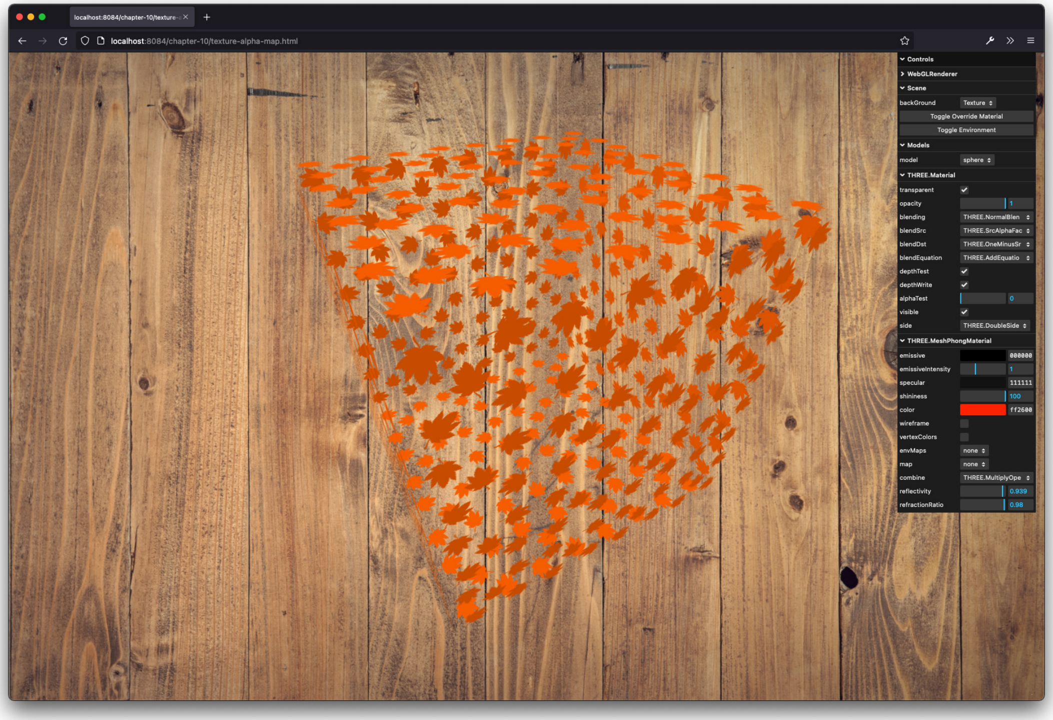
```
const metalnessTexture = new THREE.TextureLoader().load(
  '/assets/textures/engraved/Engraved_Metal_003_ROUGH.jpg',
  (texture) => {
    texture.wrapS = THREE.RepeatWrapping
    texture.wrapT = THREE.RepeatWrapping
    texture.repeat.set(4, 4)
  }
)
```

```
const material = new THREE.MeshStandardMaterial({ color:
  0xffffff })
material.metalnessMap = metalnessTexture
...
const roughnessTexture = new THREE.TextureLoader().load(
  '/assets/textures/marble/marble_0008_roughness_2k.jpg',
  (texture) => {
    texture.wrapS = THREE.RepeatWrapping
    texture.wrapT = THREE.RepeatWrapping
    texture.repeat.set(2, 2)
  }
)
const material = new THREE.MeshStandardMaterial({ color:
  0xffffff })
material.roughnessMap = roughnessTexture
```

Next up is the alpha map. With the alpha map, we can use a texture to change the transparency of parts of the model.

Using an alpha map to create transparent models

An alpha map is a way to control the opacity of the surface. If the value of the map is black, that part of the model will be fully transparent, and if it is white, it will be fully opaque. Before we look at the texture and how to apply it, we'll first look at the example (`texture-alpha-map.html`):


localhost:8084/chapter-10/texture-a
localhost:8084/chapter-10/texture-alpha-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshPhongMaterial

backGround	Texture
model	sphere
transparent	☑
opacity	1
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.DoubleSide
emissive	000000
emissiveIntensity	1
specular	111111
shininess	100
color	ff2600
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	THREE.MultiplyOpe
reflectivity	0.939
refractionRatio	0.98

Figure 10.16 – Alpha map used to give partial transparency

In this example, we've rendered a cube and set the `alphaMap` property of the material. If you open this example, make sure to set the `transparency` property of the material to `true`. You'll probably notice that you can only see the front-facing part of the cube, unlike the preceding screenshot, where you can look through the cube and see the other side. The reason is that, by default, the side property of the used material is set to `THREE.FrontSide`. To render the side that is normally hidden, we have to set the side property of the material to `THREE.DoubleSide`; you will see that the cube is rendered as shown in the preceding screenshot.

The texture that we used in this example is a very simple one:

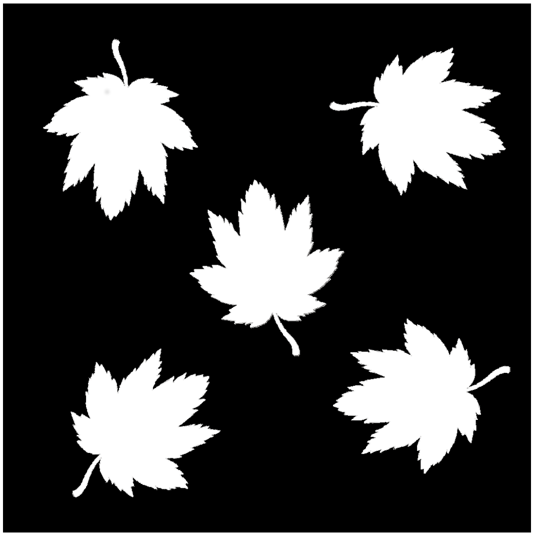


Figure 10.17 – Texture used to create a transparent model

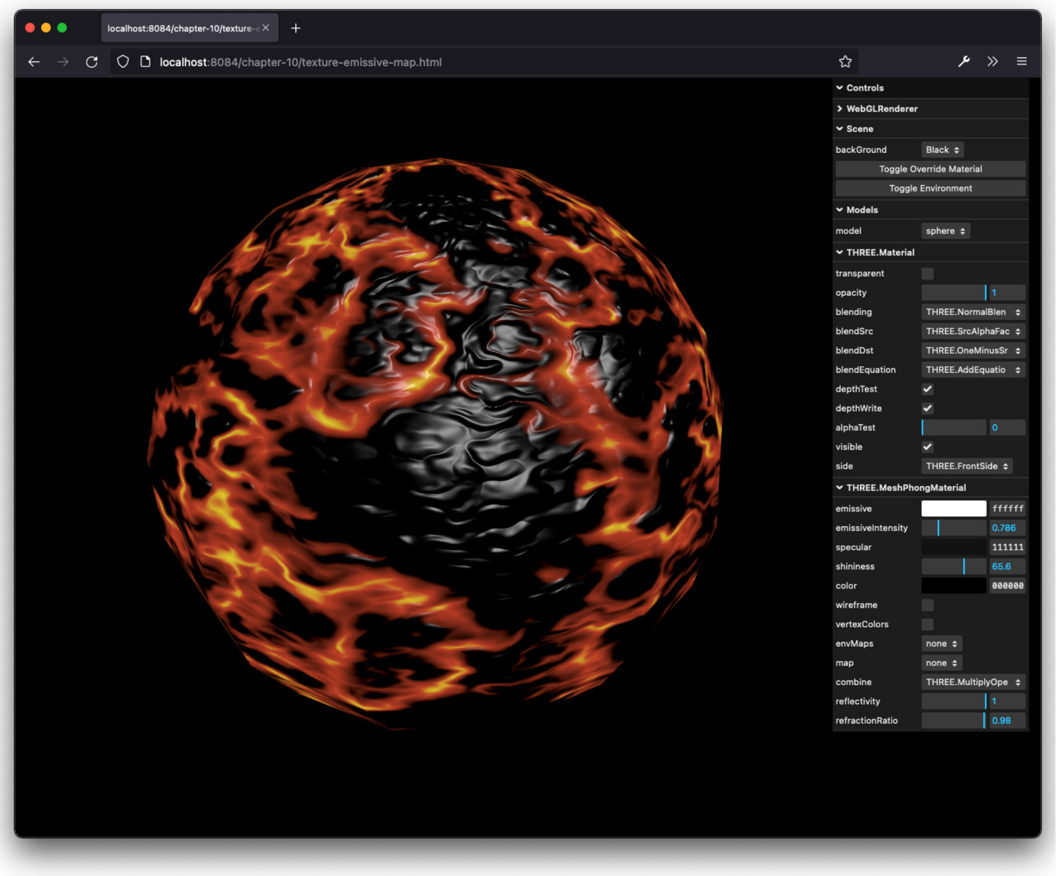
To load it, we must use the same approach as that of the other textures:

```
const alphaMap = new THREE.TextureLoader().load('/assets/
  textures/alpha/partial-transparency.png', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.alphaMap = alphaMap
material.transparent = true
```

In this code fragment, you can also see that we've set the `wrapS`, `wrapT`, and `repeat` properties of the texture. We'll explain these properties in more detail later in this chapter, but these properties can be used to determine how often we want to repeat the texture on the mesh. If set to `(1, 1)`, the whole texture won't be repeated when applied to the mesh; if set to higher values, the texture will shrink and will be repeated multiple times. In this case, we repeated it in both directions four times.

Using an emissive map for models that glow

The emissive map is a texture that can be used to make certain parts of the model glow, just like the `emissive` property does for the whole model. Just as for the `emissive` property, using an emissive map doesn't mean this object is emitting light – it just makes the part of the model where this texture is applied seem to glow. This is easier to understand by looking at an example. If you open up the `texture-emissive-map.html` example in your browser, you will see a lava-like object:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-emissive-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshPhongMaterial

backGround Black
model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive ffffff
emissiveIntensity 0.786
specular 111111
shininess 65.6
color 000000
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98

Figure 10.18 – Lava-like object using an emissive map

When you look closely, though, you might see that while the objects seem to glow, the objects themselves don't emit light. This means that you can use this to enhance objects, but the objects themselves don't contribute to the lighting of the scene. For this example, we used an emissive map that looks as follows:

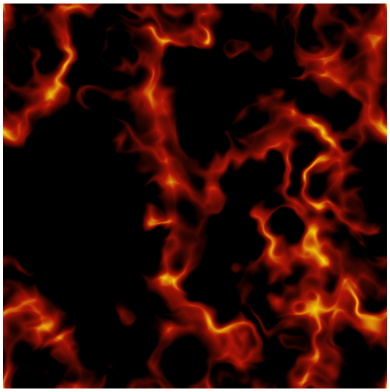


Figure 10.19 – Lava texture

To load and use an emissive map, we can use a `THREE.TextureLoader` to load one and assign it to the `emissiveMap` property (together with some other maps to get the model shown in *Figure 10.18*):

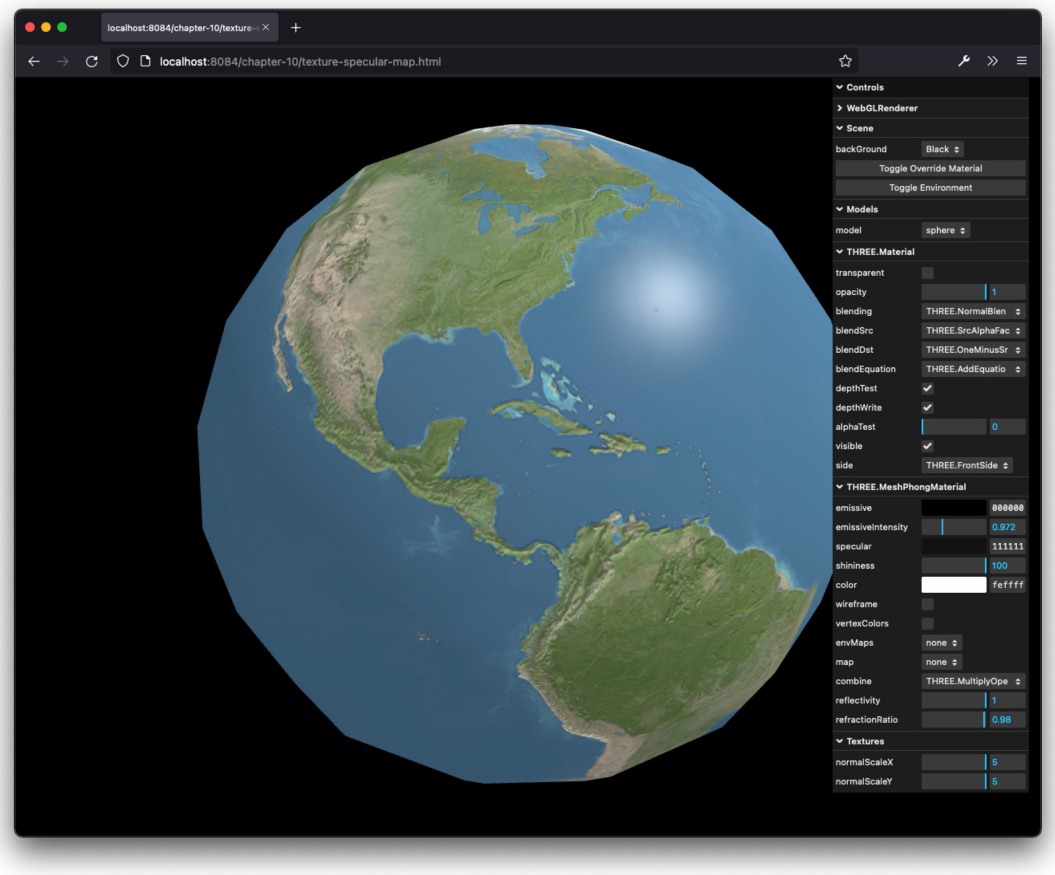
```
const emissiveMap = new   THREE.TextureLoader().load
  ('/assets/textures/lava/lava.png', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
const roughnessMap = new THREE.TextureLoader().load
  ('/assets/textures/lava/lava-smoothness.png', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
```

```
const normalMap = new THREE.TextureLoader().load
  ('/assets/textures/lava/lava-normals.png', (texture) => {
  texture.wrapS = THREE.RepeatWrapping
  texture.wrapT = THREE.RepeatWrapping
  texture.repeat.set(4, 4)
})
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.normalMap = normalMap
material.roughnessMap = roughnessMap
material.emissiveMap = emissiveMap
material.emissive = new THREE.Color(0xffffff)
material.color = new THREE.Color(0x000000)
```

Since the color from `emissiveMap` is modulated with the emissive property, make sure that you set the `emissive` property of the material to something other than black.

Using a specular map to determine shininess

In the previous examples, we mostly used `THREE.MeshStandardMaterial`, and the different maps supported by that material. `THREE.MeshStandardMaterial` is often your best choice if you need a material since it can be easily configured to represent a large number of different types of real-world materials. In older versions of Three.js, you had to use `THREE.MeshPhongMaterial` for shiny materials and `THREE.MeshLambertMaterial` for non-shiny materials. The specular map used in this section can only be used together with `THREE.MeshPhongMaterial`. With a specular map, you can define which parts of the model should be shiny, and which parts of them should be rough (similar to `metalnessMap` and `roughnessMap`, which we saw earlier). In the `texture-specular-map.html` example, we've rendered the Earth and used a specular map to make the oceans shinier than the landmasses:


localhost:8084/chapter-10/texture-s
localhost:8084/chapter-10/texture-specular-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

backGround Black
model sphere
transparent ☐
opacity 1
blending THREE.NormalBlen
blendSrc THREE.SrcAlphaFac
blendDst THREE.OneMinusSr
blendEquation THREE.AddEquatio
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 0.972
specular 111111
shininess 100
color feffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOpe
reflectivity 1
refractionRatio 0.98
normalScaleX 5
normalScaleY 5

Figure 10.20 – Specular map showing reflecting oceans

By using the menu at the top right, you can play around with the specular color and the shininess. As you can see, these two properties affect how the oceans reflect light, but they don't change the landmasses' shininess. This is because we've used the following specular map:

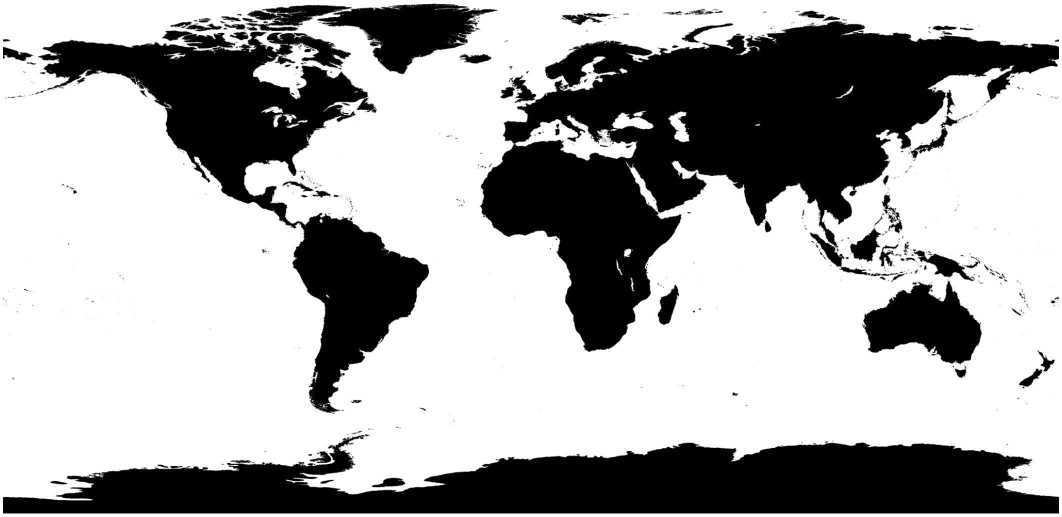


Figure 10.21 – Specular map texture

In this map, the black color means that those parts of the map have a shininess of 0%, and the white parts have a shininess of 100%.

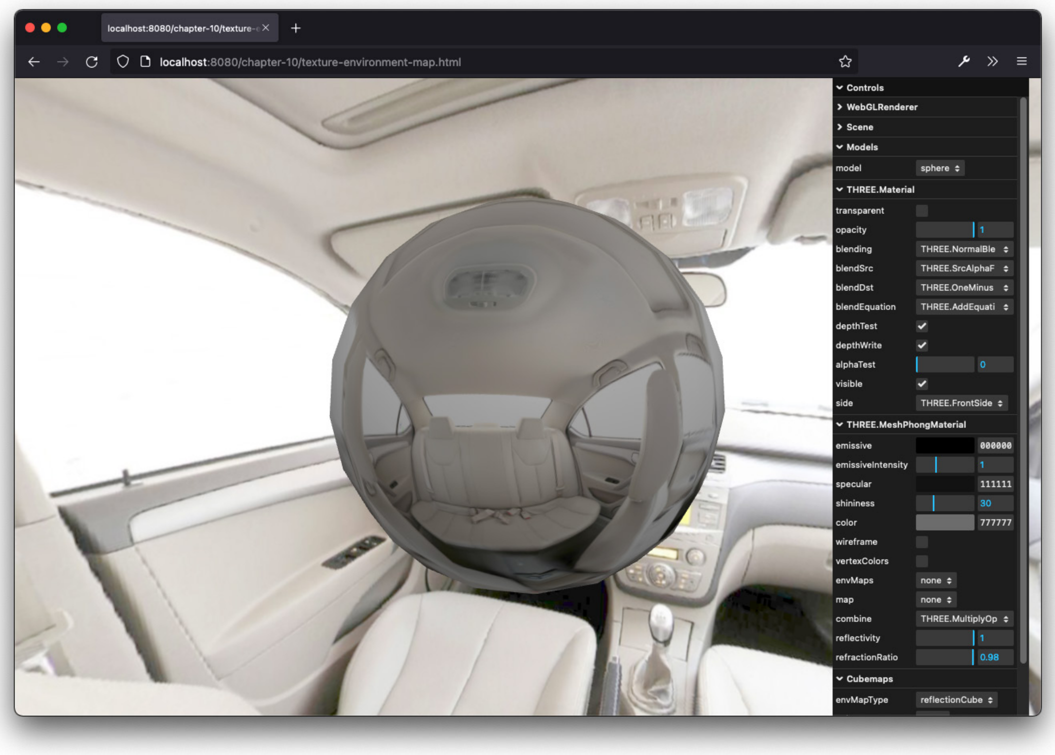
To use a specular map, we must use `THREE.TextureLoader` to load the map and assign it to the `specularMap` property of a `THREE.MathPhongMaterial`:

```
const colorMap = new THREE.TextureLoader().load
  ('/assets/textures/specular/Earth.png')
const specularMap = new THREE.TextureLoader().load
  ('/assets/textures/specular/EarthSpec.png')
const normalMap = new THREE.TextureLoader().load
  ('/assets/textures/specular/EarthNormal.png')
const material = new THREE.MeshPhongMaterial({ color:
  0xffffff })
material.map = colorMap
material.specularMap = specularMap
material.normalMap = normalMap
```

With the specular map, we've discussed most of the basic textures that you can use to add depth, color, transparency, or additional light effects to your model. In the next two sections, we'll look at one more type of map, which will allow you to add environment reflections to your model.

Creating fake reflections using an environment map

Calculating environment reflections is very CPU-intensive, and it usually requires a ray tracer approach. If you want to use reflections in Three.js, you can still do that, but you'll have to fake it. You can do so by creating a texture of the environment the object is in and applying it to the specific object. First, we'll show you the result that we're aiming for (see `texture-environment-map.html`, which is shown in the following screenshot):


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-environment-map.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshPhongMaterial
Cubemaps

model sphere
transparent ☐
opacity 1
blending THREE.NormalBle
blendSrc THREE.SrcAlphaF
blendDst THREE.OneMinus
blendEquation THREE.AddEquati
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 1
specular 111111
shininess 30
color 777777
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOp
reflectivity 1
refractionRatio 0.98
envMapType reflectionCube

Figure 10.22 – Environment map showing the inside of a car

In the preceding screenshot, you can see that the sphere reflects the environment. If you move your mouse around, you will also see that the reflection corresponds with the camera angle, concerning the environment you see. To create this example, perform the following steps:

1. Create a `CubeTexture` object. A `CubeTexture` is a set of six textures that can be applied to each side of a cube.
2. Set the skybox. When we have a `CubeTexture`, we can set it as the background of the scene. If we do this, we effectively create a very large box, inside of which the cameras and objects are placed, so that when we move the camera around, the background of the scene also changes correctly. Alternatively, we could also create a very large cube, apply `CubeTexture`, and add it to the scene ourselves.
3. Set the `CubeTexture` object as a texture of the `cubeMap` property of the material. The same `CubeTexture` object that we used to simulate the environment should be used as a texture on the meshes. Three.js will make sure it looks like a reflection of the environment.

Creating a `CubeTexture` is pretty easy, once you have the source material. What you will need are six images that, together, make up a complete environment. So, you will need the following pictures:

- Looking forward (`posz`)
- Looking backward (`negz`)
- Looking up (`posy`)
- Looking down (`negy`)
- Looking right (`posx`)
- Looking left (`negx`)

Three.js will patch these together to create a seamless environment map. There are several sites where you can download panoramic images, but they are often in a spherical equirectangular format, which looks as follows:

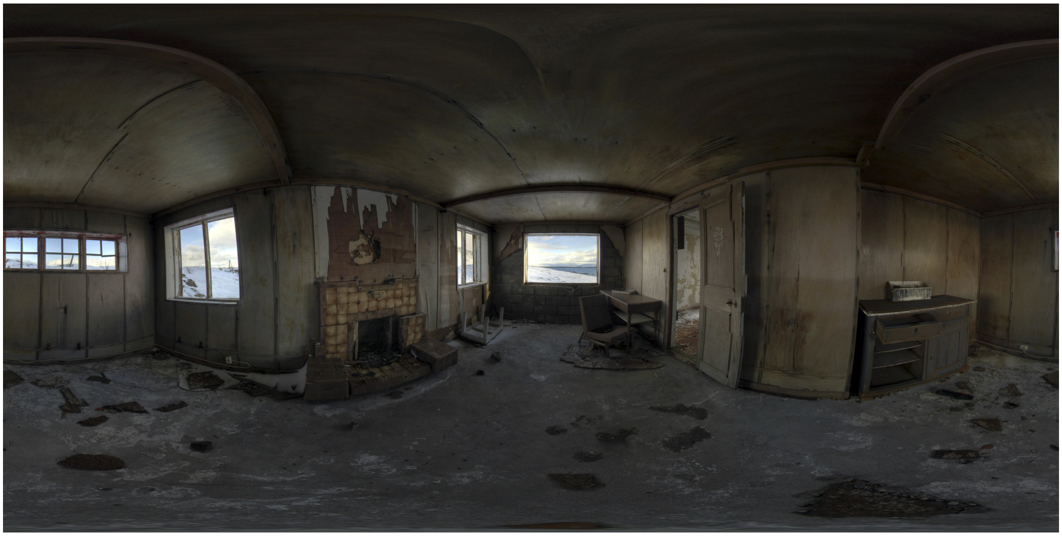


Figure 10.23 – Equirectangular format cube map

There are two ways you can use these kinds of maps. First, you can convert it into a cube map format consisting of six separate files. You can convert this online using the following site: `https://jaxry.github.io/panorama-to-cubemap/`.

Alternatively, you can use a different way to load this texture into Three.js, which we'll show later in this section.

To load a `CubeTexture` from six separate files, we can use `THREE.CubeTextureLoader`, like this:

```
const cubeMapFlowers = new THREE.CubeTextureLoader().load([
  '/assets/textures/cubemap/flowers/right.png',
  '/assets/textures/cubemap/flowers/left.png',
  '/assets/textures/cubemap/flowers/top.png',
  '/assets/textures/cubemap/flowers/bottom.png',
  '/assets/textures/cubemap/flowers/front.png',
  '/assets/textures/cubemap/flowers/back.png'
])
const material = new THREE.MeshPhongMaterial({ color:
  0x777777 }
material.envMap = cubeMapFlowers
material.mapping = THREE.CubeReflectionMapping
```

Here, you can see that we've loaded a `cubeMap` out of several different images. Once loaded, we assign the texture to the `envMap` property of a material. Finally, we must inform Three.js of which kind of mapping we want to use. If you load a texture using `THREE.CubeTextureLoader`, you can use `THREE.CubeReflectionMapping` or `THREE.CubeRefractionMapping`. The first one will make your object show reflections based on the loaded `cubeMap`, while the second one will turn your model into a more translucent glass-like object that refracts the lights slightly, once again based on the information from `cubeMap`.

We can also set this `cubeMap` as a background for the scene, like this:

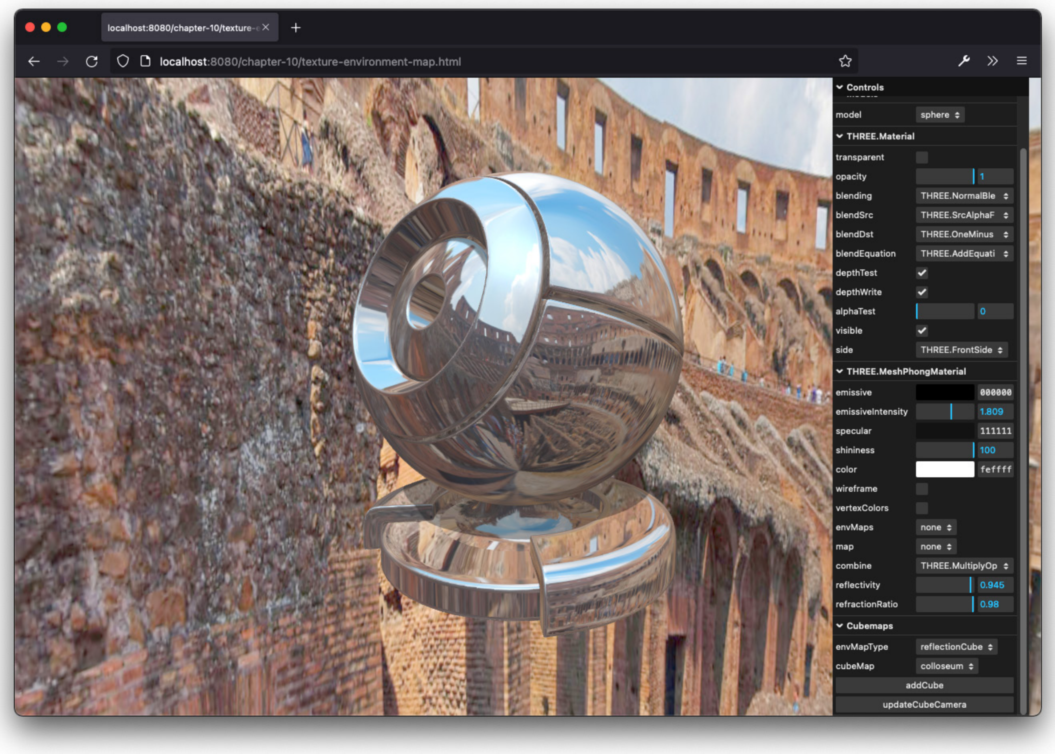
```
scene.background = cubeMapFlowers
```

When you've got a single image, the process isn't much different:

```
const cubeMapEqui = new THREE.TextureLoader().load
  ('/assets/equi.jpeg')
const material = new THREE.MeshPhongMaterial({ color:
  0x777777 }
material.envMap = cubeMapEqui
material.mapping = THREE.EquirectangularReflectionMapping
scene.background = cubeMapFlowers
```

This time, we used the normal texture loader, but by specifying a different `mapping`, we can inform Three.js how to render this texture. When using this approach, you can set the mapping to either `THREE.EquirectangularRefractionMapping` or `THREE.EquirectangularReflectionMapping`.

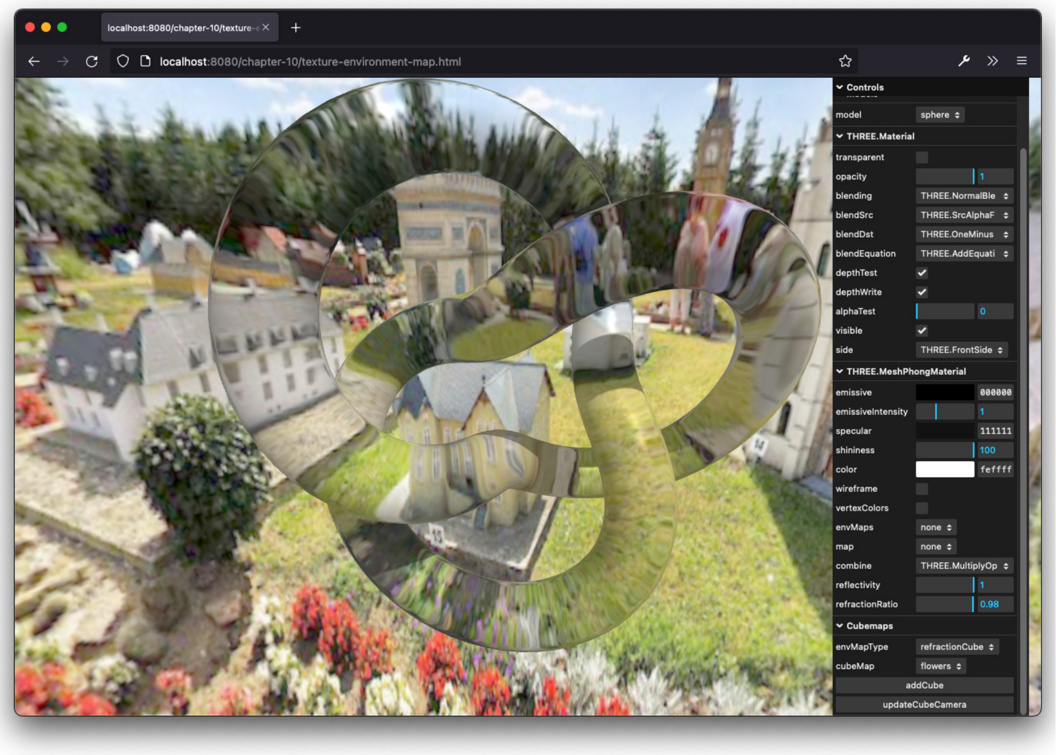
The result of both these approaches is a scene where it looks like we're standing in a wide, outdoor environment, where the meshes reflect the environment. The menu on the side allows you to set the properties of the material:


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-environment-map.html
Controls
addCube
updateCubeCamera

model	sphere
THREE.Material	
transparent	☐
opacity	1
blending	THREE.NormalBle
blendSrc	THREE.SrcAlphaF
blendDst	THREE.OneMinus
blendEquation	THREE.AddEquati
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide
THREE.MeshPhongMaterial	
emissive	000000
emissiveIntensity	1.809
specular	111111
shininess	100
color	feffff
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	THREE.MultiplyOp
reflectivity	0.945
refractionRatio	0.98
Cubemaps	
envMapType	reflectionCube
cubeMap	colloseum

Figure 10.24 – Using refraction to create glass-like objects

Aside from reflection, Three.js also allows you to use a `cubeMap` object for refraction (glass-like objects). The following screenshot shows this (you can test this yourself by using the menu on the right):


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-environment-map.html
addCube
updateCubeCamera

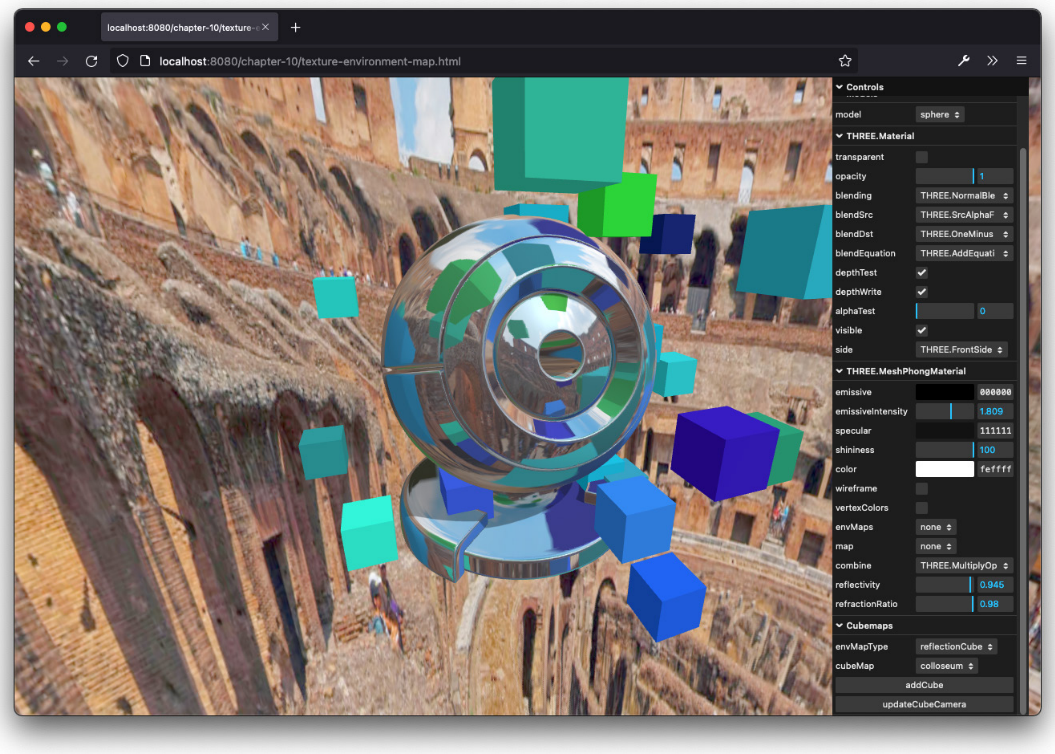
Controls	
model	sphere
THREE.Material	
transparent	☐
opacity	1
blending	THREE.NormalBle
blendSrc	THREE.SrcAlphaF
blendDst	THREE.OneMinus
blendEquation	THREE.AddEquati
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide
THREE.MeshPhongMaterial	
emissive	000000
emissiveIntensity	1
specular	111111
shininess	100
color	feffff
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	THREE.MultiplyOp
reflectivity	1
refractionRatio	0.98
Cubemaps	
envMapType	refractionCube
cubeMap	flowers

Figure 10.25 – Using refraction to create glass-like objects

To get this effect, we only need to set the mapping property of `cubeMap` to `THREE.CubeRefractionMapping` (the default is the reflection, which can also be set manually by specifying `THREE.CubeReflectionMapping`):

```
cubeMap.mapping = THREE.CubeRefractionMapping
```

In this example, we used a static environment map for the meshes. In other words, we only saw the environment's reflection and not the other meshes in the environment. In the following screenshot, you can see that, with a little bit of work, we can show the reflections of other objects as well:


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-environment-map.html
Controls
THREE.Material
THREE.MeshPhongMaterial
Cubemaps
addCube
updateCubeCamera

model sphere
transparent ☐
opacity 1
blending THREE.NormalBle
blendSrc THREE.SrcAlphaF
blendDst THREE.OneMinus
blendEquation THREE.AddEquati
depthTest ☑
depthWrite ☑
alphaTest 0
visible ☑
side THREE.FrontSide
emissive 000000
emissiveIntensity 1.809
specular 111111
shininess 100
color feffff
wireframe ☐
vertexColors ☐
envMaps none
map none
combine THREE.MultiplyOp
reflectivity 0.945
refractionRatio 0.98
envMapType reflectionCube
cubeMap colloseum

Figure 10.26 – Using a cubeCamera to create dynamic reflections

To also show reflections from the other objects in the scene, we need to use some other Three.js components. The first of them is an additional camera called `THREE.CubeCamera`:

```
const cubeRenderTarget = new THREE.WebGLCubeRenderTarget
  (128, {
  generateMipmaps: true,
  minFilter: THREE.LinearMipmapLinearFilter
})
const cubeCamera = new THREE.CubeCamera(0.1, 10,
cubeRenderTarget)
cubeCamera.position.copy(mesh.position);
scene.add(cubeCamera);
```

We will use `THREE.CubeCamera` to take a snapshot of the scene with all of the objects rendered and use that to set up a `cubeMap`. The first two arguments define the near and far properties of the camera. So, in this case, the camera only renders what it can see from 0.1 to 1.0. The last property is the target to which we want to render the texture. For that, we've created an instance of a `THREE.WebGLCubeRenderTarget`. The first parameter is the size of the render target. The higher the value, the more detailed the reflection will look. The other two properties are used to determine how the texture is scaled up and down when you zoom in.

You need to make sure that you position this camera at the exact location of `THREE.Mesh` on which you want to show the dynamic reflections. In this example, we copied the position from the mesh so that the camera is positioned correctly.

Now that we have `CubeCamera` set up correctly, we need to make sure that what `CubeCamera` sees is applied as a texture to the cube in our example. To do this, we must set the `envMap` property to `cubeCamera.renderTarget`:

```
cubeMaterial.envMap = cubeRenderTarget.texture;
```

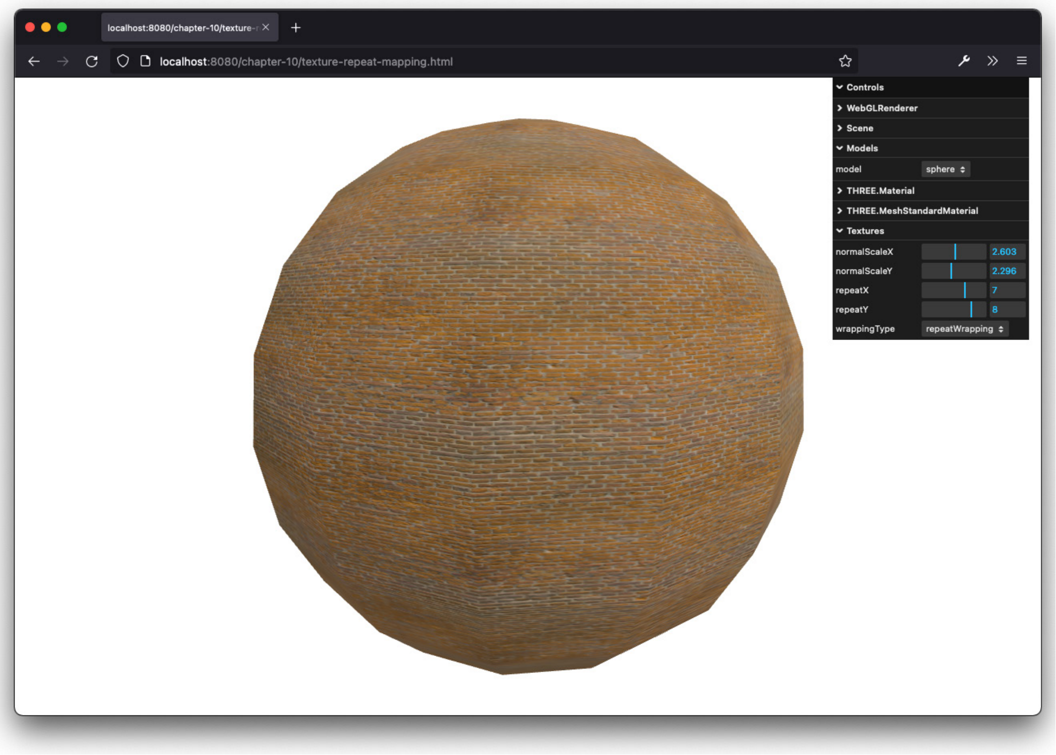
Now, we have to make sure that `cubeCamera` renders the scene so that we can use that output as input for the cube. For this, we must update the render loop as follows (or if the scene doesn't change, we can just call this once):

```
const render = () => {
...
mesh.visible = false;
cubeCamera.update(renderer, scene);
mesh.visible = true;
requestAnimationFrame(render);
renderer.render(scene, camera);
....
}
```

As you can see, first, we disable the visibility of `mesh`. We do this because we only want to see reflections from the other objects. Next, we render the scene using `cubeCamera` by calling the `update` function. After that, we make `mesh` visible again and render the scene as normal. The result is that, in the reflection of `mesh`, you can see the cubes that we added. For this example, every time you click on the `updateCubeCamera` button, the `envMap` property of the mesh will be updated.

Repeat wrapping

When you apply a texture to a geometry created by Three.js, Three.js will try to apply the texture as optimally as possible. For instance, for cubes, this means that each side will show the complete texture, and for spheres, the complete texture is wrapped around the sphere. However, there are situations where you won't want the texture to spread around a complete face or the complete geometry, but rather have the texture repeat itself. Three.js provides functionality that allows you to control this. An example where you can play around with the repeat properties is provided in `texture-repeat-mapping.html`. The following screenshot shows this example:



localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-repeat-mapping.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshStandardMaterial
Textures

model sphere
normalScaleX 2.603
normalScaleY 2.296
repeatX 7
repeatY 8
wrappingType repeatWrapping

Figure 10.27 – Repeat wrapping on a sphere

Before this property has the desired effect, you need to make sure that you set the wrapping of the texture to `THREE.RepeatWrapping`, as shown in the following code snippet:

```
mesh.material.map.wrapS = THREE.RepeatWrapping;
mesh.material.map.wrapT = THREE.RepeatWrapping;
```

The `wrapS` property defines how you want the texture to wrap along its *X*-axis, and the `wrapT` property defines how the texture should be wrapped along its *Y*-axis. Three.js provides three options for this, which are as follows:

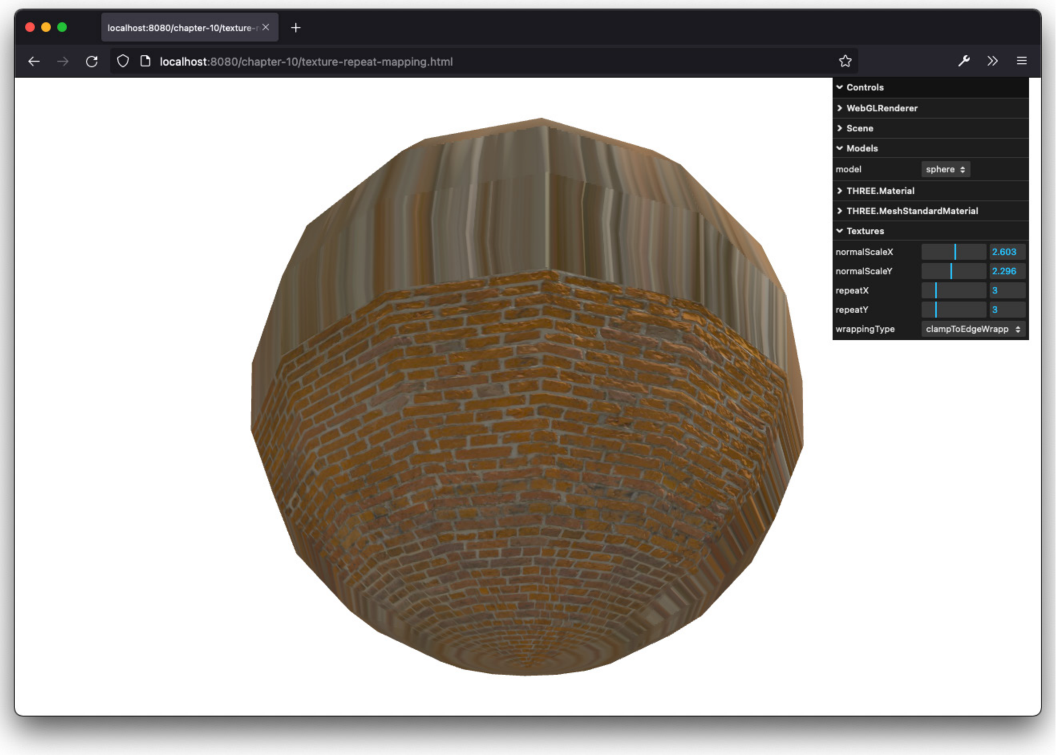
- `THREE.RepeatWrapping` allows the texture to repeat itself
- `THREE.MirroredRepeatWrapping` allows the texture to repeat itself, but each repetition is mirrored
- `THREE.ClampToEdgeWrapping` is a default setting where the texture doesn't repeat as a whole; only the pixels at the edge are repeated

In this example, you can play around with the various repeat settings and `wrapS` and `wrapT` options. Once the wrapping type has been selected, we can set the `repeat` property, as shown in the following code fragment:

```
mesh.material.map.repeat.set(repeatX, repeatY);
```

The `repeatX` variable defines how often the texture is repeated along its *X*-axis, and the `repeatY` variable defines the same for the *Y*-axis. If these values are set to 1, the texture won't repeat itself; if they are set to a higher value, you'll see that the texture will start to repeat. You can also use values lower than 1. In that case, you'll zoom in on the texture. If you set the repeat value to a negative value, the texture will be mirrored.

When you change the `repeat` property, Three.js will automatically update the textures and render them with this new setting. If you change from `THREE.RepeatWrapping` to `THREE.ClampToEdgeWrapping`, you will have to explicitly update the texture using `mesh.material.map.needsUpdate = true;`:


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-repeat-mapping.html
Controls
WebGLRenderer
Scene
Models
THREE.Material
THREE.MeshStandardMaterial
Textures

model: sphere
normalScaleX: 2.603
normalScaleY: 2.296
repeatX: 3
repeatY: 3
wrappingType: clampToEdgeWrapp

Figure 10.28 – Clamp to edge wrapping on a sphere

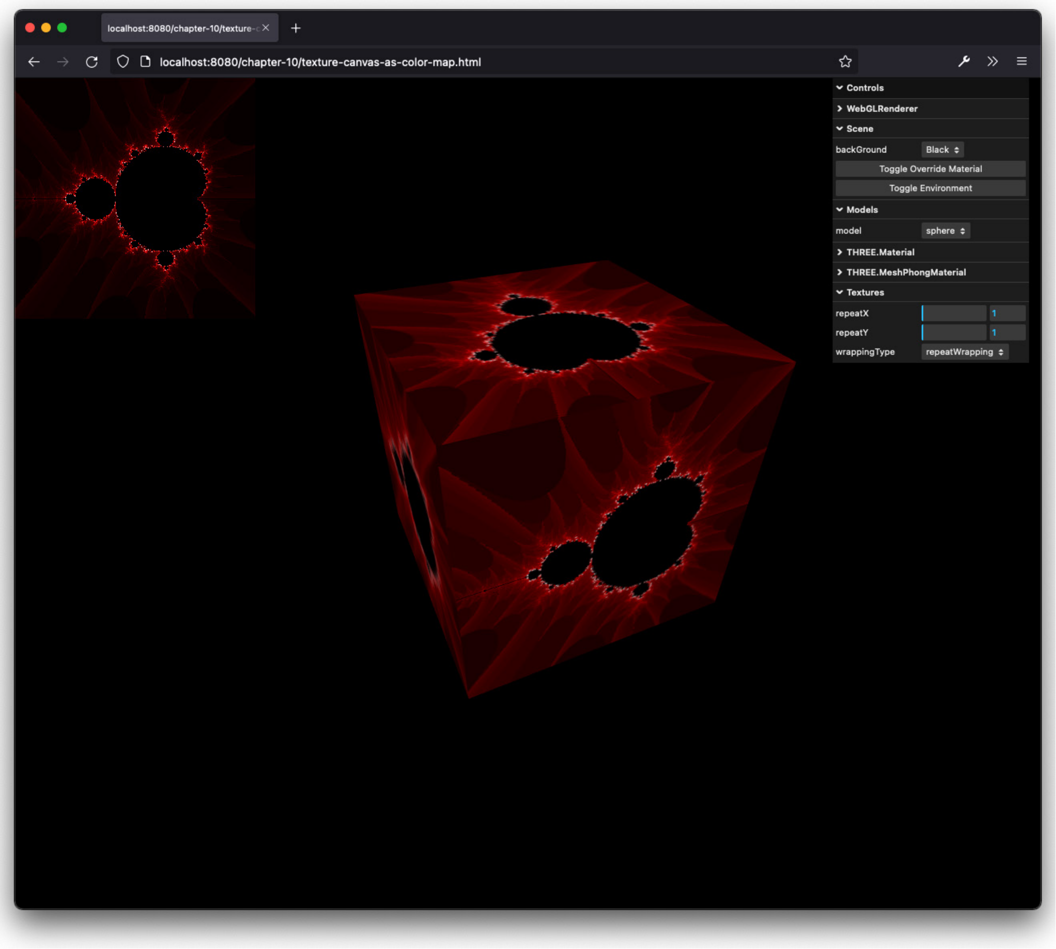
So far, we've only used static images for our textures. However, Three.js also has the option to use the HTML5 canvas as a texture.

Rendering to a canvas and using it as a texture

In this section, we will look at two different examples. First, we will look at how you can use the canvas to create a simple texture and apply it to a mesh; after that, we'll go one step further and create a canvas that can be used as a bump map, using a randomly generated pattern.

Using the canvas as a color map

In this first example, we'll render a fractal to an HTML `Canvas` element and use that as a color map for our mesh. The following screenshot shows this example (`texture-canvas-as-color-map.html`):


localhost:8080/chapter-10/texture-
localhost:8080/chapter-10/texture-canvas-as-color-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

backGround Black

model sphere

repeatX 1

repeatY 1

wrappingType repeatWrapping

Figure 10.29 – Using an HTML canvas as a texture

First, we'll look at the code required to render the fractal:

```
import Mandelbrot from 'mandelbrot-canvas'
...
const div = document.createElement('div')
div.id = 'mandelbrot'
div.style = 'position: absolute'
document.body.append(div)
const mandelbrot = new Mandelbrot(document.
getElementById('mandelbrot'), {
  height: 300,
  width: 300,
  magnification: 100
})
mandelbrot.render()
```

We won't go into too much detail, but this library requires a `div` element as input and will create a `canvas` element inside that `div`. The preceding code will render the fractal, as you can see in the previous screenshot. Next, we need to assign this canvas to the `map` property of our material:

```
const material = new THREE.MeshPhongMaterial({
  color: 0xffffff,
  map: new THREE.Texture(document.querySelector
    ('#mandelbrot canvas'))
})
material.map.needsUpdate = true
```

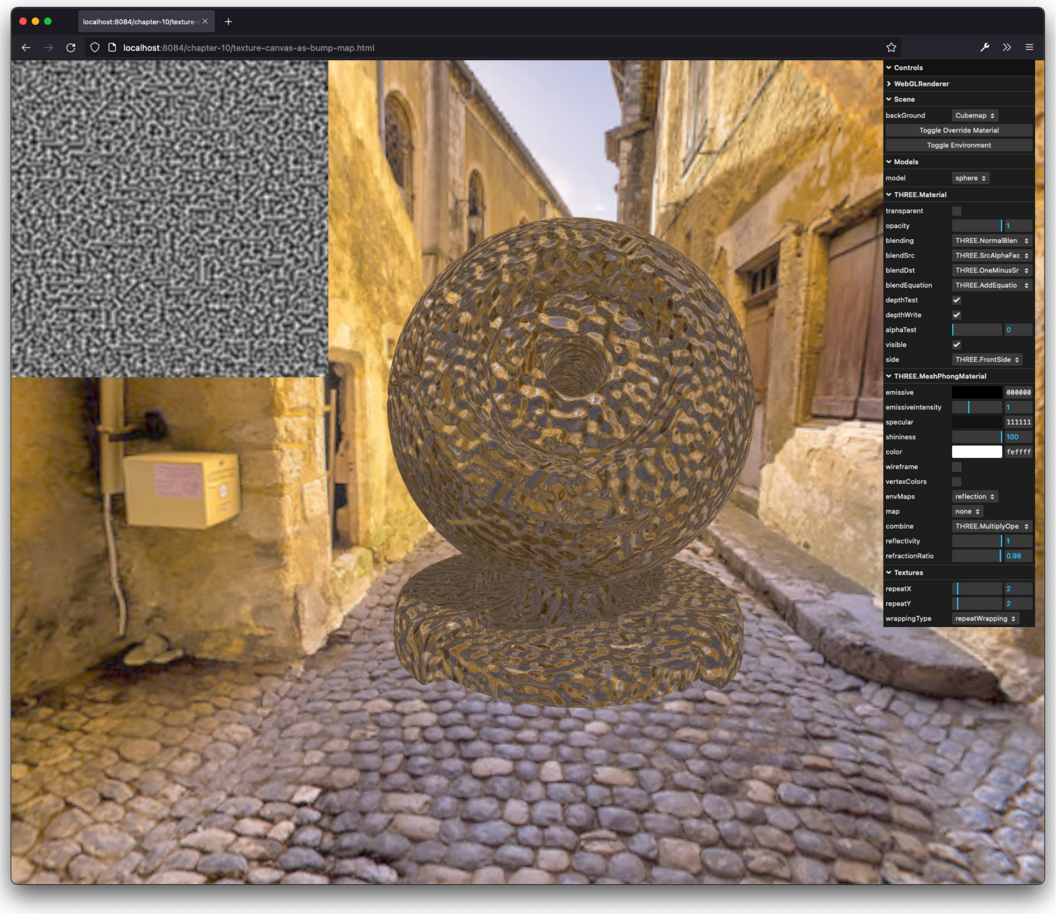
Here, we just create a new `THREE.Texture` and pass in the reference to the `canvas` element. The only thing we need to do is set `material.map.needsUpdate` to `true`, which will trigger Three.js to get the latest information from the `canvas` element, at which point we will see it applied to the mesh.

We can, of course, use this same idea for all of the different types of maps that we've seen so far. In the next example, we'll use the canvas as a bump map.

Using the canvas as a bump map

As you saw earlier in this chapter, we can add height to our model using a bump map. The higher the intensity of a pixel in this map, the higher the wrinkling. Since a bump map is just a simple black-and-white image, nothing keeps us from creating it on a canvas and using that canvas as an input for the bump map.

In the following example, we will use a canvas to generate a Perlin noise-based grayscale image, and we will use that image as input for the bump map that we apply to the cube. See the `texture-canvas-as-bump-map.html` example. The following screenshot shows this example:


localhost:8084/chapter-10/texture-
localhost:8084/chapter-10/texture-canvas-as-bump-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshPhongMaterial
Textures

backGround	Cubemap
model	sphere
transparent	☐
opacity	1
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide
emissive	000000
emissiveIntensity	1
specular	111111
shininess	100
color	feffff
wireframe	☐
vertexColors	☐
envMaps	reflection
map	none
combine	THREE.MultiplyOpe
reflectivity	1
refractionRatio	0.98
repeatX	2
repeatY	2
wrappingType	repeatWrapping

Figure 10.30 – Using an HTML canvas as a bump map

The approach for this is pretty much the same as we saw in the previous canvas example. We need to create a canvas element and fill that canvas with some noise. To do so, we must use Perlin noise. Perlin noise generates a very natural-looking texture, as you can see in the preceding screenshot. More information on Perlin noise and other noise generators can be found here: `https://thebookofshaders.com/11/`. The code to accomplish this is shown here:

```
import generator from 'perlin'
var canvas = document.createElement('canvas')
canvas.className = 'myClass'
const size = 512
canvas.style = 'position:absolute;'
canvas.width = size
canvas.height = size
document.body.append(canvas)
const ctx = canvas.getContext('2d')
for (var x = 0; x < size; x++) {
  for (var y = 0; y < size; y++) {
    var base = new THREE.Color(0xffffff)
    var value = (generator.noise.perlin2(x / 8, y / 8) + 1) / 2
    base.multiplyScalar(value)
    ctx.fillStyle = '#' + base.getHexString()
    ctx.fillRect(x, y, 1, 1)
  }
}
```

We use the `generator.noise.perlin2` function to create a value from 0 to 1, based on the `x` and `y` coordinates of the `canvas` element. This value is used to draw a single pixel on the `canvas` element. Doing this for all the pixels creates the random map that you can see in the top-left corner of the preceding screenshot. This map can then be used as a bump map:

```
const material = new THREE.MeshPhongMaterial({
  color: 0xffffff,
  bumpMap: new THREE.Texture(canvas)
})
material.bumpMap.needsUpdate = true
```

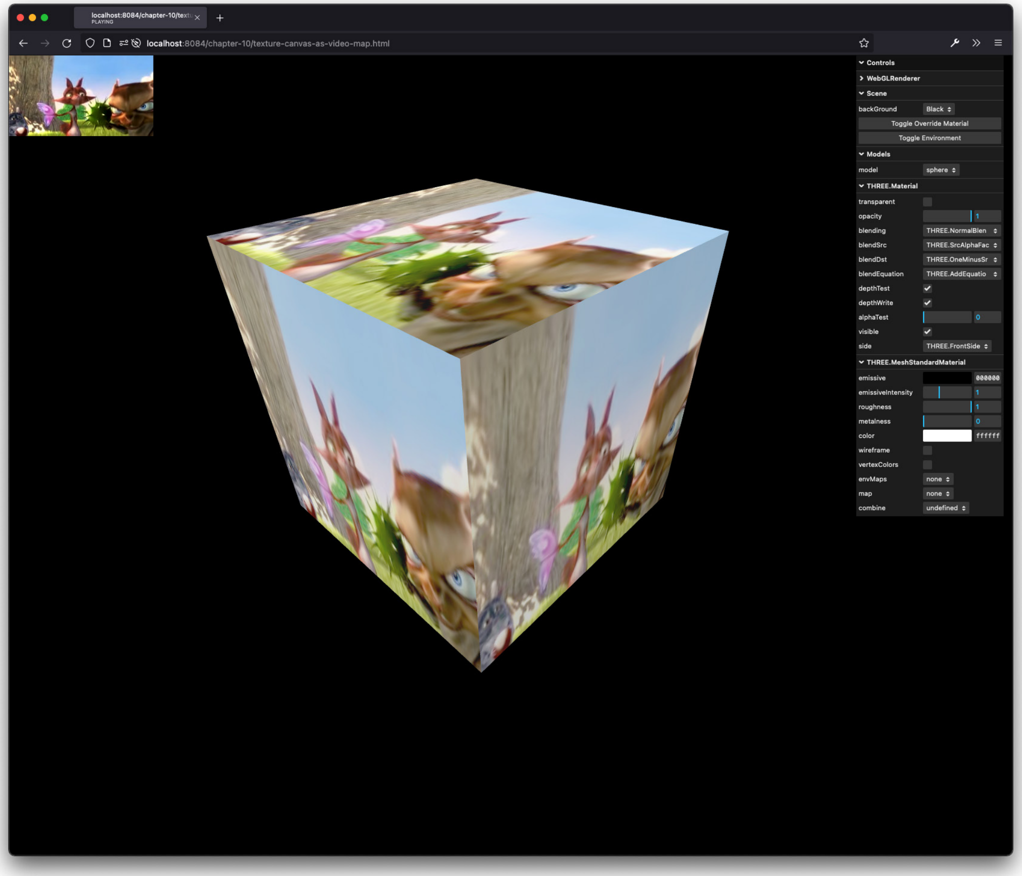
Using a THREE.DataTexture for a dynamic texture

In this example, we rendered Perlin noise using an HTML `canvas` element. Three.js also provides an alternative way to dynamically create a texture: you can create a `THREE.DataTexture` texture, where you can pass in a `Uint8Array` where you can directly set the RGB values. More information on how to use a `THREE.DataTexture` can be found here: `https://threejs.org/docs/#api/en/textures/DataTexture`.

The final input that we use for the texture is another HTML element: the HTML5 video element.

Using the output from a video as a texture

If you read the preceding section on rendering to a canvas, you may have thought about rendering video to a canvas and using that as input for a texture. That's one way to do it, but Three.js already has direct support to use the HTML5 video element (through WebGL). Check out `texture-canvas-as-video-map.html`:


localhost:8084/chapter-10/text
PLAYING
localhost:8084/chapter-10/texture-canvas-as-video-map.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Models
THREE.Material
THREE.MeshStandardMaterial

backGround	Black
model	sphere
transparent	☐
opacity	1
blending	THREE.NormalBlen
blendSrc	THREE.SrcAlphaFac
blendDst	THREE.OneMinusSr
blendEquation	THREE.AddEquatio
depthTest	☑
depthWrite	☑
alphaTest	0
visible	☑
side	THREE.FrontSide
emissive	000000
emissiveIntensity	1
roughness	1
metalness	0
color	ffffff
wireframe	☐
vertexColors	☐
envMaps	none
map	none
combine	undefined

Figure 10.31 – Using an HTML video as a texture

Using video as input for a texture is easy, just like using the canvas element. First, we need a video element to play the video:

```
const videoString = `
<video
  id="video"
  src="/assets/movies/Big_Buck_Bunny_small.ogv"
  controls="true"
</video>
`
const div = document.createElement('div')
div.style = 'position: absolute'
document.body.append(div)
div.innerHTML = videoString
```

This creates a basic HTML5 `video` element, by setting the HTML string directly to the `innerHTML` property of the `div` element. While this works great for testing, frameworks and libraries usually provide better options for this. Next, we can configure Three.js to use the video as an input for a texture, as follows:

```
const video = document.getElementById('video')
const texture = new THREE.VideoTexture(video)
const material = new THREE.MeshStandardMaterial({
  color: 0xffffff,
  map: texture
})
```

The result can be seen in the `texture-canvas-as-video-map.html` example.

Summary

With that, we have completed this chapter on textures. As you've seen, a lot of textures are available in Three.js, each with a different use. You can use any image in PNG, JPG, GIF, TGA, DDS, PVR, TGA, KTX, EXR, or RGBE format as a texture. Loading these images is done asynchronously, so remember to either use a rendering loop or add a callback when you load a texture. With the different types of textures available, you can create great-looking objects from low-poly models.

With Three.js, it is also easy to create dynamic textures, using either the HTML5 `canvas` element or the `video` element – just define a texture with these elements as the input, and set the `needsUpdate` property to `true` whenever you want the texture to be updated.

With this chapter out of the way, we've pretty much covered all of the important concepts of Three.js. However, we haven't looked at an interesting feature that Three.js offers: postprocessing. With postprocessing, you can add effects to your scene after it has been rendered. You can, for instance, blur or colorize your scene, or add a TV-like effect using scan lines. In *Chapter 11*, *Render Postprocessing*, we'll look at postprocessing and how you can apply it to your scene.

Part 4:

Post-Processing, Physics, and Sounds

In this final part, we'll look at a couple of more advanced topics. We'll explain how you can set up a post-processing pipeline, which can be used to add different kinds of effects to the final rendered scene. We'll also introduce the Rapier physics engine, and explain how you can use Three.js and Blender together. We end this part with information on how Three.js can be used together with React, TypeScript, and the Web-XR standard.

In this part, there are the following chapters:

- *Chapter 11*, *Render Postprocessing*
- *Chapter 12, Adding Physics and Sounds to Your Scene*
- *Chapter 13*, *Working with Blender and Three.js*
- *Chapter 14*, *Three.js Together with React, Typescript, and Web-XR*

11

Render Postprocessing

In this chapter, we'll look at one of the main features of Three.js that we haven't touched upon yet: render postprocessing. With render postprocessing, you can add additional effects to your scene after it is rendered. For instance, you could add an effect that makes the scene look like it is shown on an old TV, or you could add blur and bloom effects.

The main points we'll discuss in this chapter are as follows:

- Setting up Three.js for postprocessing
- Some basic postprocessing passes provided by Three.js, such as `BloomPass` and `FilmPass`
- Applying effects to part of a scene using masks
- Using `ShaderPass` to add even more basic postprocessing effects, such as sepia filters, mirror effects, and color adjustments
- Using `ShaderPass` for various blurring effects and more advanced filters
- Creating a custom postprocessing effect by writing a simple shader

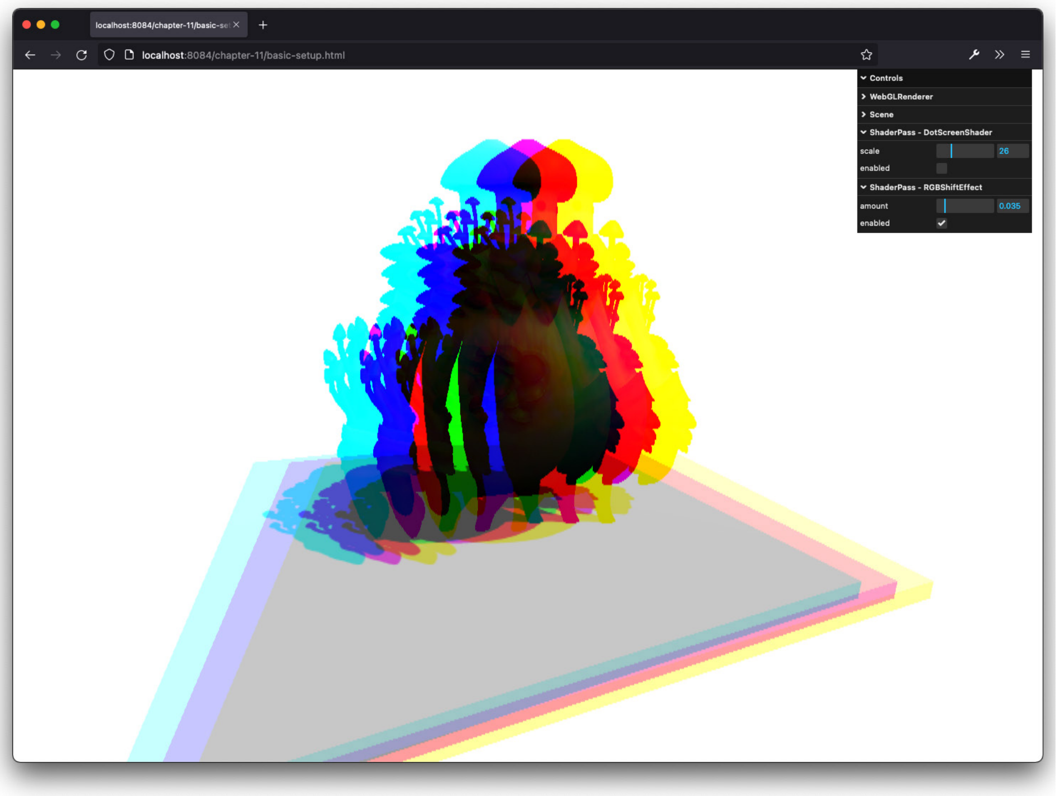
In *Chapter 1*, *Creating Your First 3D Scene with Three.js*, in the *Introducing requestAnimationFrame* section, we set up a rendering loop that we've used throughout the book, in order to render and animate our scenes. For postprocessing, we need to make a couple of changes to this setup to allow Three.js to postprocess the final rendering. In the first section, we'll look at how to do this.

Setting up Three.js for postprocessing

To set up Three.js for postprocessing, we have to make a couple of changes to our current setup, as follows:

1. Create `EffectComposer`, which can be used to add postprocessing passes.
2. Configure `EffectComposer` so that it can render our scene and apply any additional postprocessing steps.
3. In the render loop, use `EffectComposer` to render the scene, apply the configured postprocessing steps, and show the output.

As always, we will show an example that you can use to experiment with and adapt for your own purposes. The first example in this chapter can be accessed from `basic-setup.html`. You can use the menu in the top-right corner to modify the properties of the postprocessing step used in this example. In this example, we will render the mushroom man from *Chapter 9*, *Animation and Moving the Camera*, and add an RGB shift effect to it, as follows:


localhost:8084/chapter-11/basic-se
localhost:8084/chapter-11/basic-setup.html
Controls
WebGLRenderer
Scene

ShaderPass - DotScreenShader
scale 26
enabled ☐
ShaderPass - RGBShiftEffect
amount 0.035
enabled ☑

Figure 11.1 – Rendered using a postprocessing pass

This effect is added after the scene is rendered by using `ShaderPass`, together with `EffectComposer`. In the menu on the right side of the screen, you can configure this effect and also enable the `DotScreenShader` effect.

In the following sections, we'll explain the individual steps from the previous list.

Creating THREE.EffectComposer

To get `EffectComposer` to work, we first need effects that we can use with it. Three.js comes with a large number of effects and shaders you can use. In this chapter, we'll show most of them, but for a complete overview, check out the following two directories on GitHub:

- Effect passes: `https://github.com/mrdoob/three.js/tree/dev/examples/jsm/postprocessing`
- Shaders: `https://github.com/mrdoob/three.js/tree/dev/examples/jsm/shaders`

To use these effects in your scene, you need to import them:

```
import { EffectComposer } from
  'three/examples/jsm/postprocessing/EffectComposer'
import { RenderPass } from
  'three/examples/jsm/postprocessing/RenderPass.js'
import { ShaderPass } from
  'three/examples/jsm/postprocessing/ShaderPass.js'
import { BloomPass } from
  'three/examples/jsm/postprocessing/BloomPass.js'
import { GlitchPass } from
  'three/examples/jsm/postprocessing/GlitchPass.js'
import { RGBShiftShader } from
  'three/examples/jsm/shaders/RGBShiftShader.js'
import { DotScreenShader } from
  'three/examples/jsm/shaders/DotScreenShader.js'
import { CopyShader } from
  'three/examples/jsm/shaders/CopyShader.js'
```

In the imports in the preceding code block, we import the main `EffectComposer` and a different number of postprocessing passes and shaders that we can use together with this `EffectComposer`. Once we've got these, setting up `EffectComposer` is done like this:

```
const composer = new EffectComposer(renderer)
```

As you can see, the only argument an effect composer takes is `renderer`. Next, we will add various passes to this composer.

Configuring THREE.EffectComposer for postprocessing

Each pass is executed in the sequence it is added to `THREE.EffectComposer`. The first pass that we add is `RenderPass`. This pass renders our scene using the camera provided but doesn't output it to the screen yet:

```
const renderPass = new RenderPass(scene, camera);
composer.addPass(renderPass);
```

With the `addPass` function, we add `RenderPass` to `EffectComposer`. The next step is to add another pass that will take the results from `RenderPass` as its input, apply its transformation, and output its result to the screen. Not all the available passes allow for this, but the passes we've used in this example do:

```
const effect1 = new ShaderPass(DotScreenShader)
effect1.uniforms['scale'].value = 10
effect1.enabled = false
const effect2 = new ShaderPass(RGBShiftShader)
effect2.uniforms['amount'].value = 0.015
effect2.enabled = false
const composer = new EffectComposer(renderer)
composer.addPass(new RenderPass(scene, camera))
composer.addPass(effect1)
composer.addPass(effect2)
```

In this example, we've added two effects to `composer`. First, the scene is rendered using `RenderPass`, then `DotScreenShader` is applied, and finally, we apply `RGBShiftShader`.

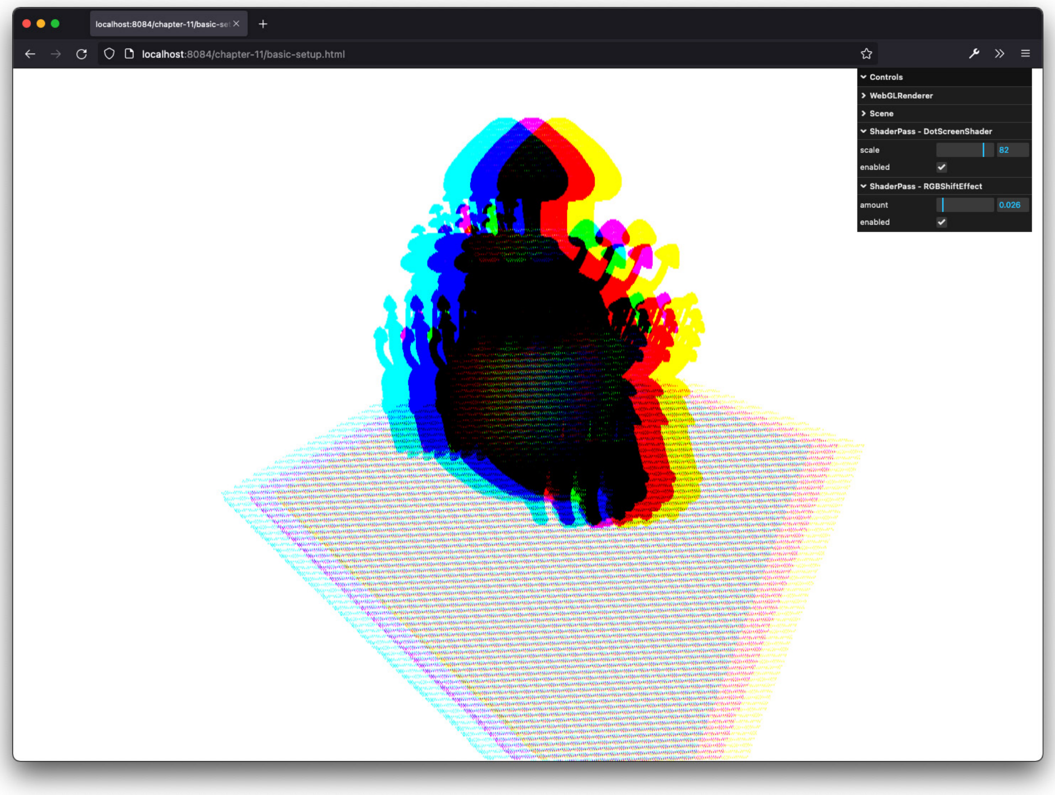
All we need to do now is update the render loop so that we render using `EffectComposer` instead of through the normal `WebGLRenderer`.

Updating the render loop

We just need to make a small modification to our render loop to use the composer instead of `THREE.WebGLRenderer`:

```
const render = () => {
requestAnimationFrame(render);
composer.render();
}
```

The only modification that we made is removing `renderer.render(scene, camera)` and replacing it with `composer.render()`. This will call the render function on `EffectComposer`, which, in turn, uses the passed-in `THREE.WebGLRenderer`, and the result is that we see the output on the screen:


localhost:8084/chapter-11/basic-se
localhost:8084/chapter-11/basic-setup.html
Controls
WebGLRenderer
Scene
ShaderPass - DotScreenShader
ShaderPass - RGBShiftEffect

scale 82
enabled ✓
amount 0.026
enabled ✓

Figure 11.2 – Rendered using multiple postprocessing passes

Using controls after applying a render pass

You can still use the normal controls to move around a scene. All of the effects that you will see in this chapter are applied after the scene is rendered. With this basic setup, we'll look at the available postprocessing passes in the next couple of sections.

Postprocessing passes

Three.js comes with a number of postprocessing passes that you can use directly with `THREE.EffectComposer`.

Use a simple GUI to experiment

Most of the shaders and passes shown in this chapter can be configured. When you want to apply one yourself, it is usually easiest to just add a simple UI that allows you to play around with the properties. That way, you can see what a good setting for your specific scenario is.

The following list shows all the postprocessing passes available in Three.js:

- `AdaptiveToneMappingPass`: This render pass adapts the luminosity of a scene, based on the amount of light available in it.
- `BloomPass`: This is an effect that makes lighter areas bleed into darker areas. This simulates an effect wherein the camera is overwhelmed by extremely bright light.
- `BokehPass`: This adds a bokeh effect to the scene. With a bokeh effect, the foreground of the scene is in focus, while the rest is out of focus.
- `ClearPass`: This spill pass clears the current texture buffer.
- `CubeTexturePass`: This can be used to render a skybox in the scene.
- `DotScreenPass`: This applies a layer of black dots, representing the original image across the screen.
- `FilmPass`: This simulates a TV screen by applying scanlines and distortions.
- `GlitchPass`: This shows an electronic glitch on the screen at a random time interval.
- `HalfTonePass`: This adds a halftone effect to the scene. With a halftone effect, the scene is rendered as a set of colored glyphs (circles, squares, and so on) of various sizes.
- `LUTPass`: With `LUTPass`, you can apply a color correction step to the scene after it is rendered (not shown in this chapter).
- `MaskPass`: This allows you to apply a mask to the current image. Subsequent passes are only applied to the masked area.
- `OutlinePass`: This renders the outline of the objects in the scene.
- `RenderPass`: This renders a scene based on the scene and camera supplied.

- `SAOPass`: This provides runtime ambient occlusion.
- `SMAAPass`: This adds an anti-aliasing effect to the scene.
- `SSAARenderPass`: This adds anti-aliasing to the scene.
- `SSAOPass`: This provides an alternative way to perform runtime ambient occlusion.
- `SSRPass`: This pass allows you to create reflective objects.
- `SavePass`: When this pass is executed, it makes a copy of the current rendering step that you can use later. This pass isn't that useful in practice, and we won't use it in any of our examples.
- `ShaderPass`: This allows you to pass in custom shaders for advanced or custom postprocessing passes.
- `TAARenderPass`: This adds an anti-aliasing effect to the scene.
- `TexturePass`: This stores the current state of the composer in a texture that you can use as input for other `EffectComposer` instances.
- `UnrealBloomPass`: This is the same as `THREE.BloomPass` but with an effect similar to the effect used in the Unreal 3D engine.

Let's start with a number of simple passes.

Simple postprocessing passes

For simple passes, we'll look at what we can do with `FilmPass`, `BloomPass`, and `DotScreenPass`. For these passes, an example is available (`multi-passes.html`) that will allow you to experiment with these passes and see how they affect the original output differently. The following screenshot shows the example:

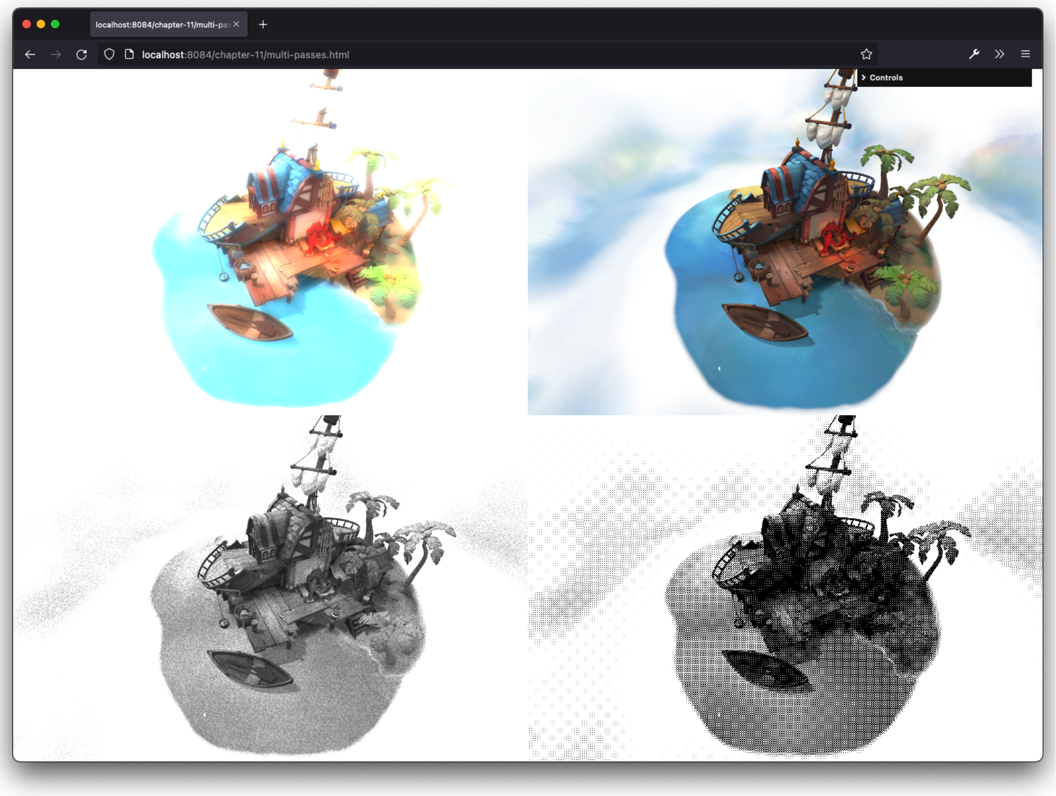

localhost:8084/chapter-11/multi-pas
localhost:8084/chapter-11/multi-passes.html
Controls

Figure 11.3 – Three simple passes applied to a scene

In this example, you can see four scenes at the same time, and in each scene, a different postprocessing pass is added. The one in the top-left corner shows `BloomPass`, the one in the bottom-right corner shows `DotScreenPass`, and the one in the bottom-left corner shows `FilmPass`. The scene in the top-right corner shows the original render.

In this example, we also use `THREE.ShaderPass` and `THREE.TexturePass` to reuse the output from the original rendering as input for the other three scenes. That way, we only need to render the scene once. So, before we look at the individual passes, let's look at these two passes, as follows:

```
const effectCopy = new ShaderPass(CopyShader)
const renderedSceneComposer = new EffectComposer(renderer)
renderedSceneComposer.addPass(new RenderPass(scene,
  camera))
renderedSceneComposer.addPass(new ShaderPass
  (GammaCorrectionShader))
```

```
renderedSceneComposer.addPass(effectCopy)
renderedSceneComposer.renderToScreen = false
const texturePass = new TexturePass
  (renderedSceneComposer.renderTarget2.texture)
```

In this piece of code, we set up `EffectComposer`, which will output the default scene (the one in the top-right corner). This composer has three passes:

- `RenderPass`: This pass renders the scene.
- `ShaderPass` with `GammaCorrectionShader`: Makes sure that the colors of the output are correct. If, after applying effects, the color of the scene looks incorrect, this shader will correct it.
- `ShaderPass` with `CopyShader`: Renders the output (without any further postprocessing to the screen, if we set the `renderToScreen` property to `true`).

If you look at the example, you can see that we show the same scene four times but with a different effect applied each time. We could also render the scene from scratch by using `RenderPass` four times, but that would be a bit of a waste, since we can just reuse the output from the first composer. To do this, we create `TexturePass` and pass in the `composer.renderTarget2.texture` value. This property contains the rendered scene as a texture, which we can pass into `TexturePass`. We can now use the `texturePass` variable as input for our other composers, without having to render the scene from scratch. Let's look at `FilmPass` first and how we can use the results from `TexturePass` as input.

Using THREE.FilmPass to create a TV-like effect

To create `FilmPass`, we use the following piece of code:

```
const filmpass = new FilmPass()
const filmpassComposer = new EffectComposer(renderer)
filmpassComposer.addPass(texturePass)
filmpassComposer.addPass(filmpass)
```

The only step that we need to take to use `TexturePass` is to add it as the first pass in our composer. Next, we can just add `FilmPass`, and the effect will be applied. `FilmPass` can take four additional parameters, as shown in the following list:

- `noiseIntensity`: This property allows you to control how grainy the scene looks.
- `scanlinesIntensity`: `FilmPass` adds a number of scanlines (see `scanLinesCount`) to the scene. With this property, you can define how prominently these scanlines are shown.

- `scanLinesCount`: The number of scanlines that are shown can be controlled with this property.
- `grayscale`: If this is set to `true`, the output will be converted to grayscale.

There are actually two ways that you can pass in these parameters. In this example, we passed them in as arguments to the constructor, but you can also set them directly, as follows:

```
effectFilm.uniforms.grayscale.value = controls.grayscale;
effectFilm.uniforms.nIntensity.value = controls.
  noiseIntensity;
effectFilm.uniforms.sIntensity.value = controls.
  scanlinesIntensity;
effectFilm.uniforms.sCount.value = controls.scanlinesCount;
```

In this approach, we use the `uniforms` property, which communicates directly with WebGL. In the *Using THREE.ShaderPass for custom effects* section, where we talk about creating a custom shader, we'll get a bit deeper into `uniforms`; for now, all you need to know is that this way, you can update the configuration of postprocessing passes and shaders and see the results directly.

The result of this pass is shown in the following figure:

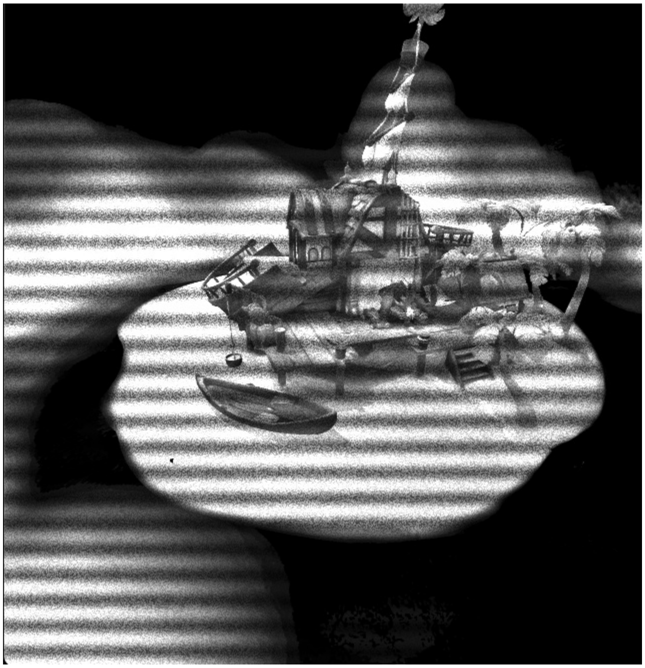


Figure 11.4 – A film effect provided by FilmPass

The next effect is the bloom effect, which you can see in the top-left part of the screen in *Figure 11.3*.

Adding a bloom effect to the scene with THREE.BloomPass

The effect that you see in the top-left corner is called the bloom effect. When you apply the bloom effect, the bright areas of a scene will be made more prominent and bleed into the darker areas. The code to create `BloomPass` is as follows:

```
const bloomPass = new BloomPass()
const effectCopy = new ShaderPass(CopyShader)
bloomPassComposer = new EffectComposer(renderer)
bloomPassComposer.addPass(texturePass)
bloomPassComposer.addPass(bloomPass)
bloomPassComposer.addPass(effectCopy)
```

If you compare this with `EffectComposer`, which we used with `FilmPass`, you'll notice that we add an additional pass, `effectCopy`. This step doesn't add any special effects but just copies the output from the last pass to the screen. We need to add this step, since `BloomPass` doesn't render directly to the screen.

The following table lists the properties that you can set on `BloomPass`:

- `strength`: This is the strength of the bloom effect. The higher this is, the more bright the brighter areas are, and the more they will bleed into the darker areas.
- `kernelSize`: This is the size of the kernel. This is the size of the area that is blurred in a single step. If you set this higher, more pixels will be included to determine the effect at a specific point.
- `sigma`: With the `sigma` property, you can control the sharpness of the bloom effect. The higher the value, the more blurred the bloom effect will look.
- `resolution`: The `resolution` property defines how precisely the bloom effect is created. If you make this too low, the result will look blocky.

A better way to understand these properties is to just experiment with them by using the aforementioned example, `multi-passes.html`. The following screenshot shows the `bloom` effect with a high sigma size and high strength:

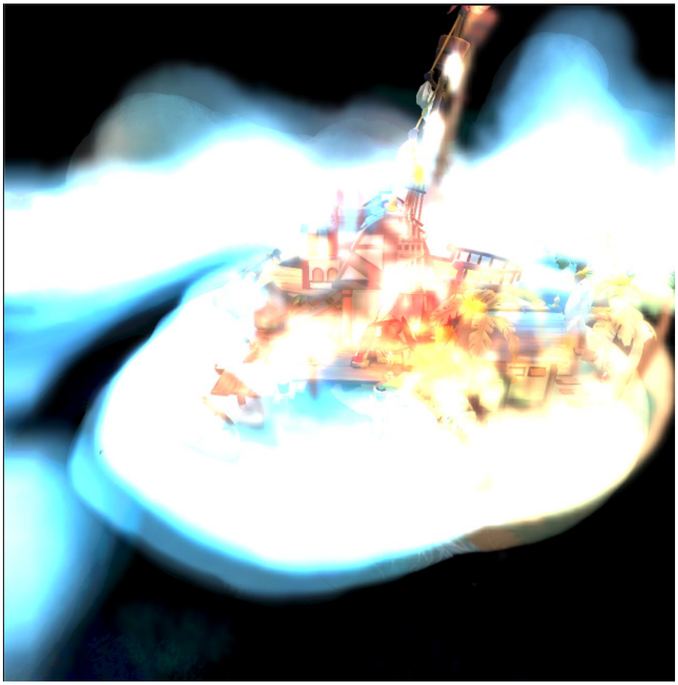


Figure 11.5 – The bloom effect using BloomPass

The next simple effects that we'll look at are the `DotScreenPass` effects.

Outputting a scene as a set of dots

Using `DotScreenPass` is very similar to using `BloomPass`. We just saw `BloomPass` in action. Let's now look at the code for `DotScreenPass`:

```
const dotScreenPass = new DotScreenPass()
const dotScreenPassComposer = new EffectComposer(renderer)
dotScreenPassComposer.addPass(texturePass)
dotScreenPassComposer.addPass(dotScreenPass)
```

With this effect, we don't need `effectCopy` to output the result to the screen.

`DotScreenPass` can also be configured with a number of properties, as follows:

- `center`: With the `center` property, you can fine-tune the way the dots are offset.
- `angle`: The dots are aligned in a certain manner. With the `angle` properties, you can change this alignment.
- `scale`: With this, we can set the sizes of the dots to use. The lower the scale, the larger the dots.

What applies to the other shaders also applies to this shader. It's much easier to get the right settings with experimentation, as seen in the following figure:

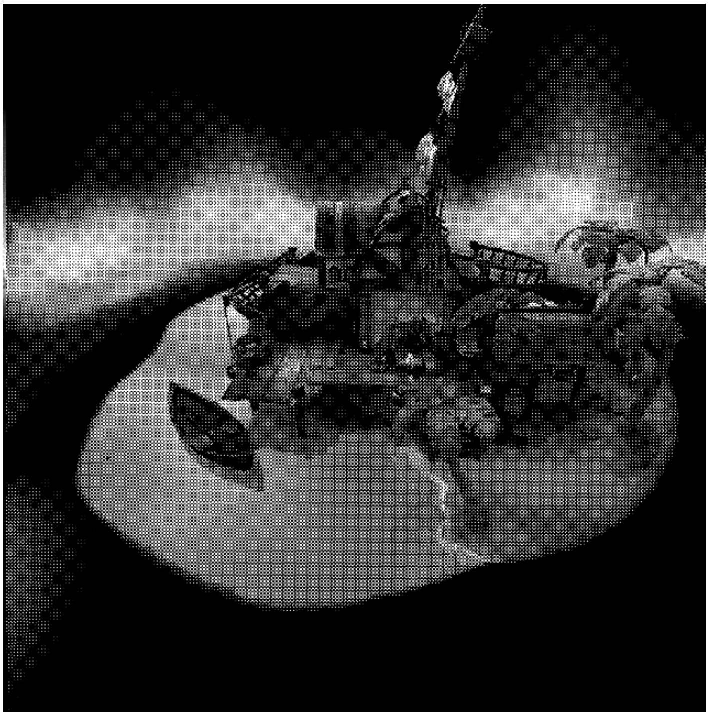


Figure 11.6 – A dot-screen effect using DotScreenPass

Before we move on to the next set of simple shaders, we'll first look at how we've rendered multiple scenes on the same screen.

Showing the output of multiple renderers on the same screen

This section won't go into detail on how to use postprocessing effects but will explain how to get the output of all four `EffectComposer` instances on the same screen. First, let's look at the render loop used for this example:

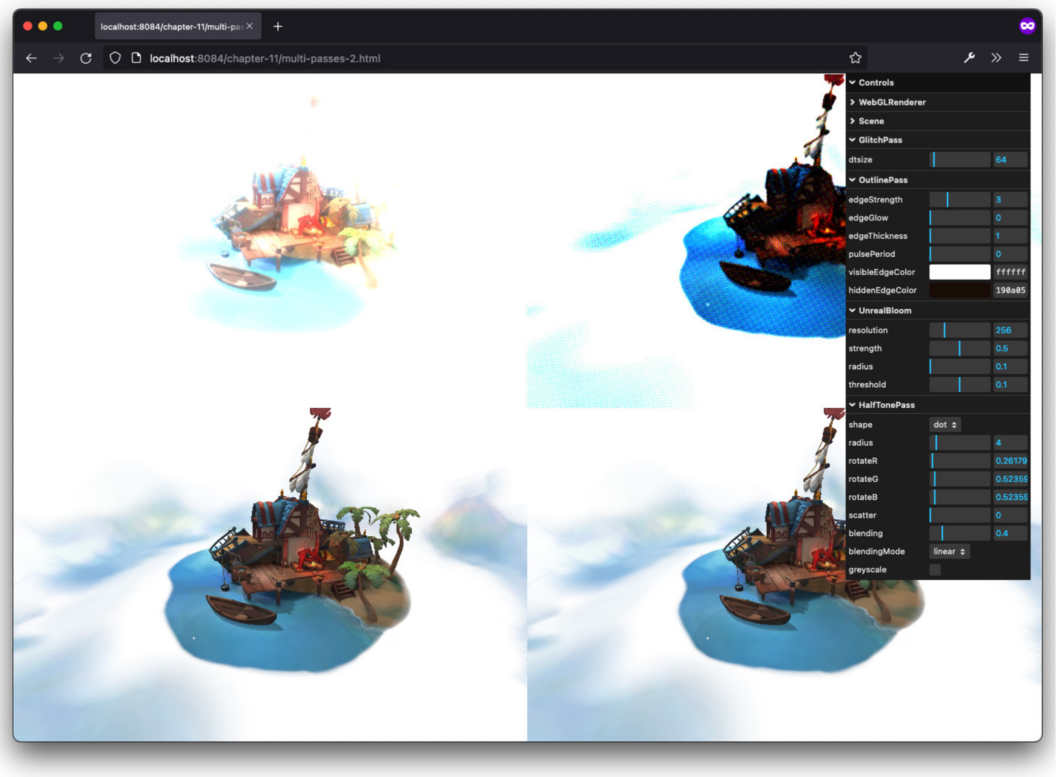
```
const width = window.innerWidth || 2
const height = window.innerHeight || 2
const halfWidth = width / 2
const halfHeight = height / 2
const render = () => {
  renderer.autoClear = false
  renderer.clear()
  renderedSceneComposer.render()
  renderer.setViewport(0, 0, halfWidth, halfHeight)
  filmpassComposer.render()
  renderer.setViewport(halfWidth, 0, halfWidth, halfHeight)
  dotScreenPassComposer.render()
  renderer.setViewport(0, halfHeight, halfWidth,
    halfHeight)
  bloomPassComposer.render()
  renderer.setViewport(halfWidth, halfHeight, halfWidth,
    halfHeight)
  copyComposer.render()
  requestAnimationFrame(() => render())
}
```

The first thing to notice is that we set the `renderer.autoClear` property to `false` and then explicitly call the `clear()` function in the render loop. If we don't do this each time we call the `render()` function on a composer, the previously rendered parts of the screen will be cleared. With this approach, we only clear everything at the beginning of our render loop.

To avoid having all of our composers render in the same space, we set the `viewport` function of the renderer, which is used by our composers, to a different part of the screen. This function takes four arguments: `x`, `y`, `width`, and `height`. As you can see in the code sample, we use this function to divide the screen into four areas and make the composers render to their individual areas. Note that you can also use this approach with multiple `scene`, `camera`, and `WebGLRenderer` instances, if you want. With this setup, the render loop will render each of the four `EffectComposer` objects to their own parts of the screen. Let's quickly look at another couple of passes.

Additional simple passes

If you open the `multi-passes-2.html` example in your browser, you will see a number of additional passes in action:


localhost:8084/chapter-11/multi-pa
localhost:8084/chapter-11/multi-passes-2.html

Controls
WebGLRenderer
Scene
GlitchPass
dtsize 64
OutlinePass
edgeStrength 3
edgeGlow 0
edgeThickness 1
pulsePeriod 0
visibleEdgeColor ffffff
hiddenEdgeColor 190a05
UnrealBloom
resolution 256
strength 0.5
radius 0.1
threshold 0.1
HalfTonePass
shape dot
radius 4
rotateR 0.26179
rotateG 0.52359
rotateB 0.52359
scatter 0
blending 0.4
blendingMode linear
greyscale

Figure 11.7 – Another set of four passes

We won't go into too much detail here, since these passes are configured in the same way as those in the previous sections. In this example, you can see the following:

- In the bottom-left corner, you can see `OutlinePass`. The outline pass can be used to draw an outline for a `THREE.Mesh` object.
- In the bottom-right corner, `GlitchPass` is shown. As the name implies, this pass provides a technical rendering glitch effect.
- In the top-left corner, the `UnrealBloom` effect is shown.
- In the top-right corner, `HalftonePass` is used to convert the rendering to a set of dots.

As is the case for all of the examples in this chapter, you can configure the individual properties of these passes by using the menu on the right.

To see `OutlinePass` correctly, you can set the scene background to black and zoom out a bit:

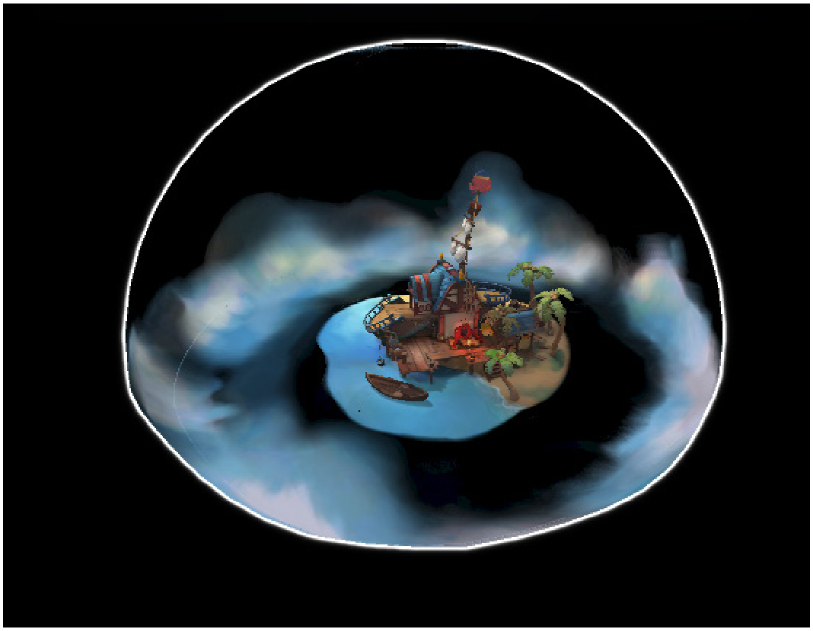


Figure 11.8 – The outline pass showing the outline of the scene

So far we've seen simple effects, in the next section, we'll look at how you can use masks to apply effects to parts of the screen.

Advanced EffectComposer flows using masks

In the previous examples, we applied the postprocessing pass to a complete screen. However, Three.js also has the ability to apply passes only to a specific area. In this section, we will perform the following steps:

1. Create a scene to serve as a background image.
2. Create a scene containing a sphere that looks like Earth.
3. Create a scene containing a sphere that looks like Mars.
4. Create `EffectComposer`, which renders these three scenes into a single image.
5. Apply a colorify effect to the sphere rendered as Mars.
6. Apply a sepia effect to the sphere rendered as Earth.

This might sound complex, but it is actually surprisingly easy to accomplish. First, let's look at the result that we're aiming for in the `masks.html` example. The following screenshot shows the results of these steps:

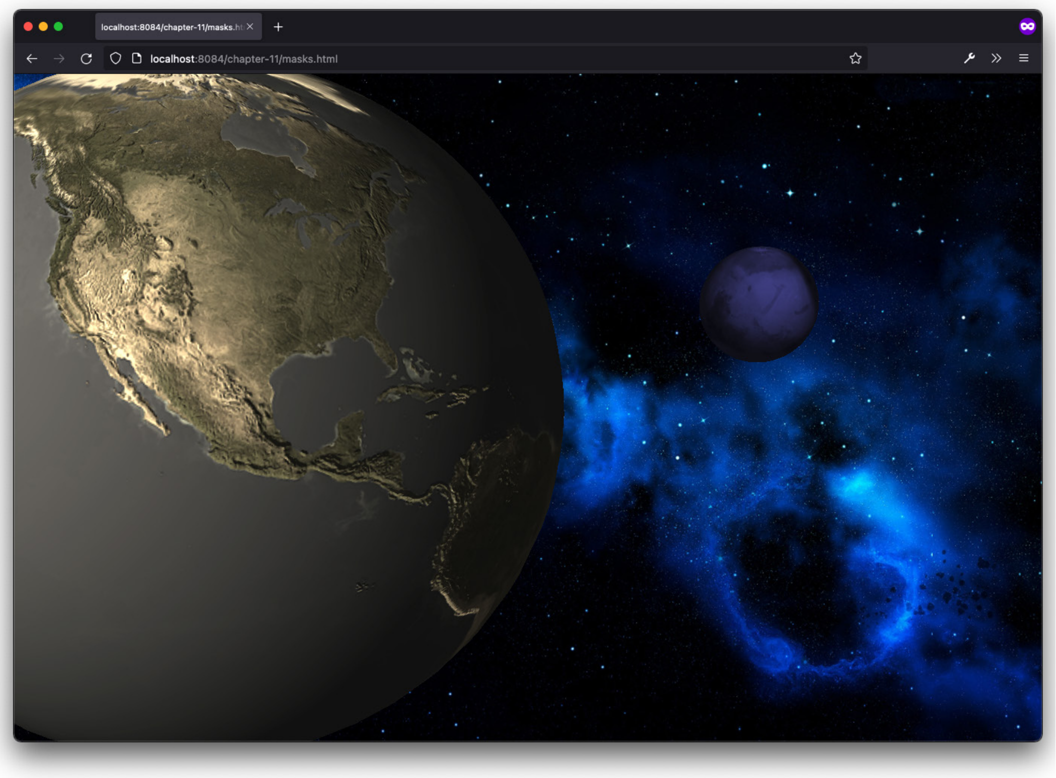

localhost:8084/chapter-11/masks.ht
localhost:8084/chapter-11/masks.html

Figure 11.9 – Use a mask to apply an effect to part of the screen

The first thing that we need to do is set up the various scenes that we'll be rendering:

```
const sceneEarth = new THREE.Scene()
const sceneMars = new THREE.Scene()
const sceneBG = new THREE.Scene()
```

To create the Earth and Mars spheres, we just create the spheres with the correct material and textures and add them to their specific scenes. For the background scene, we load a texture and set it to as the background for `sceneBG`. This is shown in the following code (`addEarth` and `addMars` are just helper functions to keep the code clear; they create a simple `THREE.Mesh` from `THREE.SphereGeometry`, create some lights, and add them all to `THREE.Scene`):

```
sceneBG.background = new THREE.TextureLoader().load
 ('/assets/textures/bg/starry-deep-outer-space-galaxy.jpg')
const earthAndLight = addEarth(sceneEarth)
sceneEarth.translateX(-16)
sceneEarth.scale.set(1.2, 1.2, 1.2)
const marsAndLight = addMars(sceneMars)
sceneMars.translateX(12)
sceneMars.translateY(6)
sceneMars.scale.set(0.2, 0.2, 0.2)
```

In this example, we use the `background` property of a scene to add the starry background. There is an alternative way to create a background. We can use `THREE.OrhoGraphicCamera`. With `THREE.OrthographicCamera`, the size of the rendered object doesn't change when it is closer or further away from the camera, so, by positioning a `THREE.PlaneGeometry` object directly in front of `THREE.rhoGraphicCamera`, we can create a background as well.

We now have got our three scenes, and we can start to set up our passes and `EffectComposer`. Let's start by looking at the complete chain of passes, after which we'll look at the individual passes:

```
var composer = new EffectComposer(renderer)
composer.renderTarget1.stencilBuffer = true
composer.renderTarget2.stencilBuffer = true
composer.addPass(bgRenderPass)
composer.addPass(earthRenderPass)
composer.addPass(marsRenderPass)
composer.addPass(marsMask)
composer.addPass(effectColorify)
composer.addPass(clearMask)
composer.addPass(earthMask)
composer.addPass(effectSepia)
composer.addPass(clearMask)
composer.addPass(effectCopy)
```

To work with masks, we need to create `EffectComposer` in a slightly different manner. We need to set the `stencilBuffer` property of the internally used render targets to `true`. A stencil buffer, a special type of buffer, is used to limit the area of rendering. So, by enabling the stencil buffer, we can use our masks. Let's look at the first three passes that are added. These three passes render the background, the Earth scene, and the Mars scene, as follows:

```
const bgRenderPass = new RenderPass(sceneBG, camera)
const earthRenderPass = new RenderPass(sceneEarth, camera)
earthRenderPass.clear = false
const marsRenderPass = new RenderPass(sceneMars, camera)
marsRenderPass.clear = false
```

There's nothing new here, except that we set the `clear` property of two of these passes to `false`. If we don't do this, we'll only see the output from the `marsRenderPass` render, since it will clear everything before it starts rendering.

If you look back at the code for `EffectComposer`, the next three passes are `marsMask`, `effectColorify`, and `clearMask`. First, we'll look at how these three passes are defined:

```
const marsMask = new MaskPass(sceneMars, camera)
const effectColorify = new ShaderPass(ColorifyShader)
effectColorify.uniforms['color'].value.setRGB(0.5, 0.5, 1)
const clearMask = new ClearMaskPass()
```

The first of these three passes is `MaskPass`. When creating a `MaskPass` object, you pass in a scene and a camera, just as you did for `RenderPass`. A `MaskPass` object will render this scene internally, but instead of showing this on screen, it will use the rendered internal scene to create a mask. When a `MaskPass` object is added to `EffectComposer`, all of the subsequent passes will be applied only to the mask defined by `MaskPass`, until a `ClearMaskPass` step is encountered. In this example, this means that the `effectColorify` pass, which adds a blue glow, is only applied to the objects rendered in `sceneMars`.

We use the same approach to apply a sepia filter to the Earth object. We first create a mask based on the Earth scene and use this mask in `EffectComposer`. After using `MaskPass`, we add the effect that we want to apply (`effectSepia` in this case), and, once we're done with that, we add `ClearMaskPass` to remove the mask again.

The last step for this specific `EffectComposer` is one that we've already seen. We need to copy the final result to the screen, and we once again use the `effectCopy` pass for that. With this setup, we can apply the effects that we want to be a part of the total screen. Be aware, though, that these effects are applied to a part of the rendered image if the Mars scene and the Earth scene overlap.

The effects of both will be applied to that part of the screen:

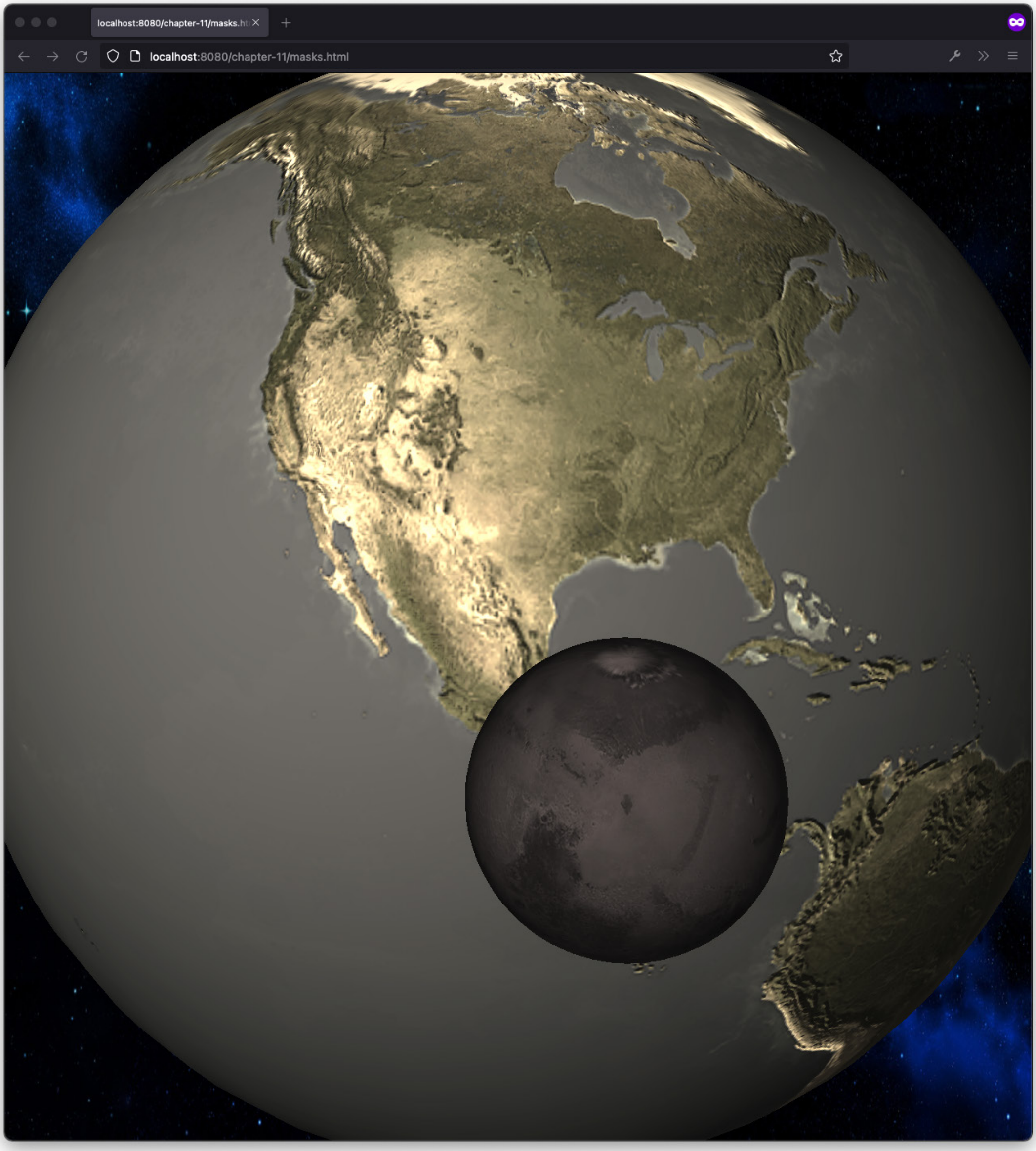

localhost:8080/chapter-11/masks.ht
localhost:8080/chapter-11/masks.html

Figure 11.10 – When masks overlap, both effects are applied

There is one additional property that's interesting when working with `MaskPass`, and that's the `inverse` property. If this property is set to `true`, the mask is inverted. In other words, the effect is applied to everything but the scene passed into `MaskPass`. This is shown in the following screenshot, where we set the `inverse` property of `earthMask` to `true`:

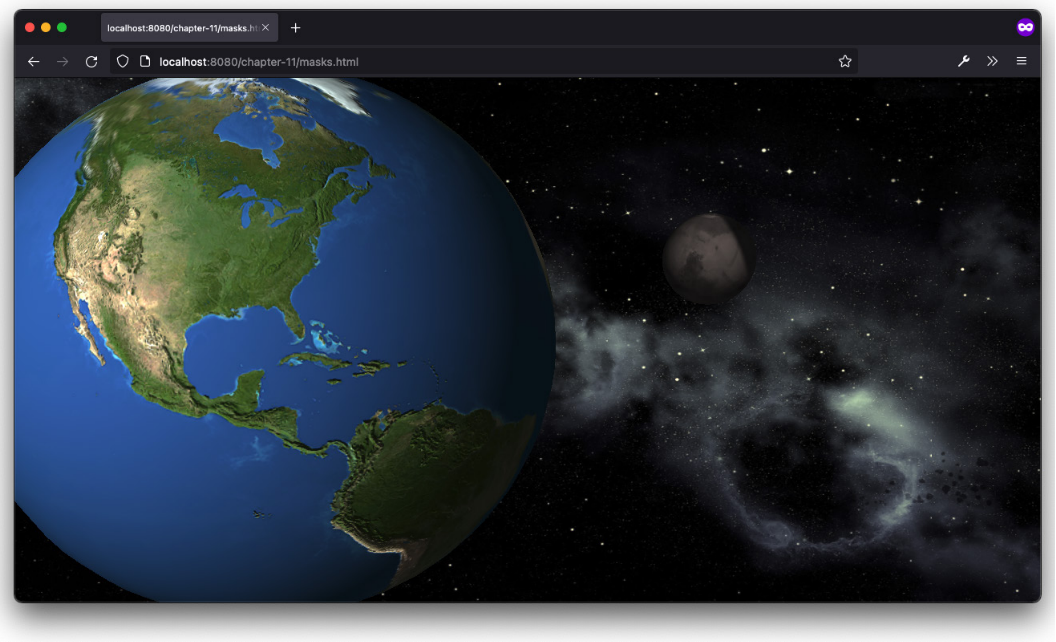
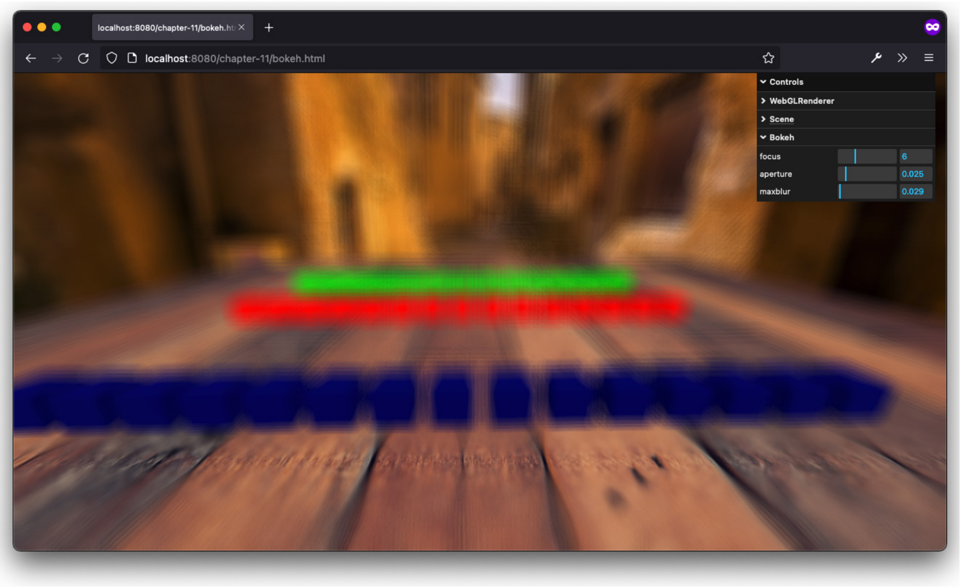

localhost:8080/chapter-11/masks.ht
localhost:8080/chapter-11/masks.html

Figure 11.11 – When masks overlap, both effects are applied

Before we move on to a discussion of `ShaderPass`, we're going to look at two passes that provide a more advanced effect: `BokehPass` and `SSAOPass`.

Advanced pass – bokeh

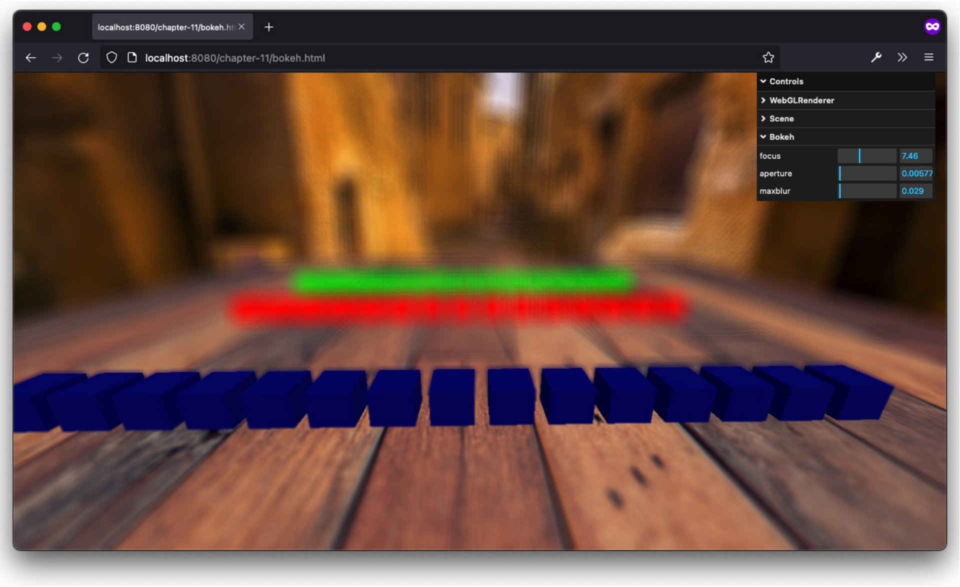
With the `BokehPass`, you can add a `bokeh` effect to your scene. In a bokeh effect, only part of the scene is in focus, and the rest of the scene looks blurry. To see this effect in action, you can open the `bokeh.html` example:


localhost:8080/chapter-11/bokeh.ht
localhost:8080/chapter-11/bokeh.html
Controls
WebGLRenderer
Scene
Bokeh

focus 6
aperture 0.025
maxblur 0.029

Figure 11.12 – An unfocussed bokeh effect

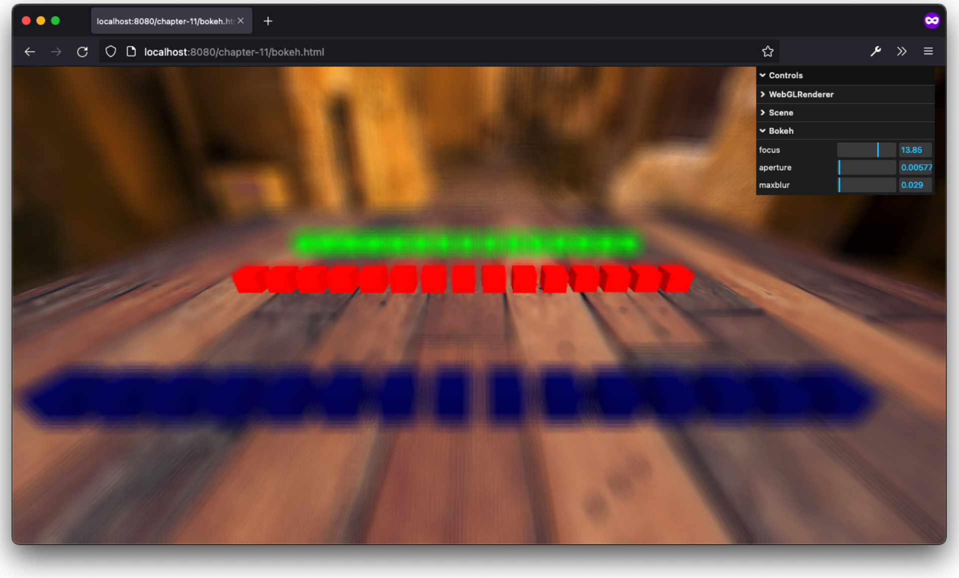
When you open it, initially, the whole scene will look blurry. With the **Bokeh** controls on the right, you can set the focus value to the part of the scene that you want to have in focus, and play around with the `aperture` property to determine the size of the area that should be in focus. By sliding the focus, you can have the set of cubes in the foreground in focus, as follows:


localhost:8080/chapter-11/bokeh.ht
localhost:8080/chapter-11/bokeh.html
Controls
WebGLRenderer
Scene
Bokeh

focus 7.46
aperture 0.00577
maxblur 0.029

Figure 11.13 – Bokeh focussed on the first set of cubes

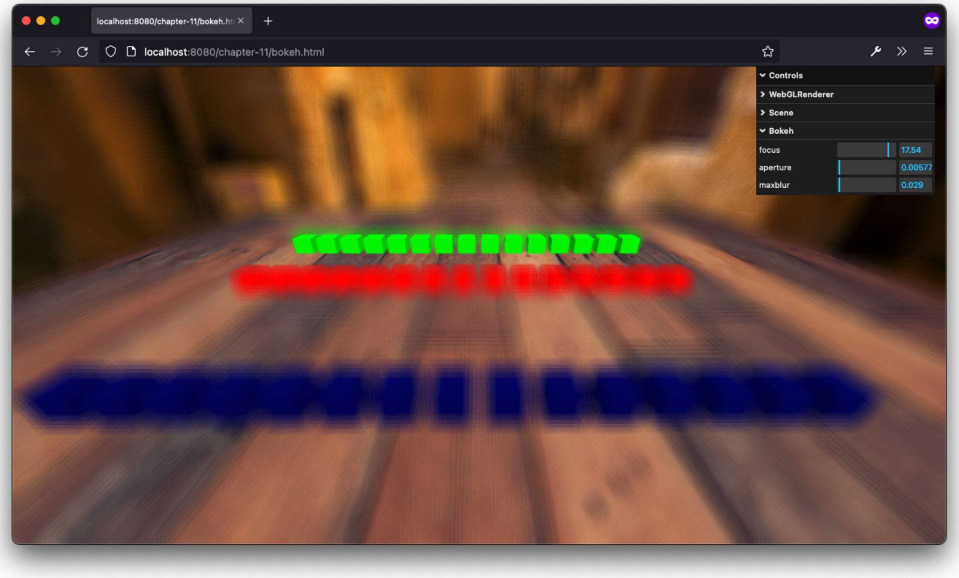
Alternatively, if we slide the focus further, we can focus on the red cubes:


localhost:8080/chapter-11/bokeh.ht
localhost:8080/chapter-11/bokeh.html
Controls
WebGLRenderer
Scene
Bokeh

focus 13.85
aperture 0.00577
maxblur 0.029

Figure 11.14 – Bokeh focussed on the second set of cubes

And, if we slide the focus even further, we can focus on the set of green cubes at the far end of the scene:


localhost:8080/chapter-11/bokeh.ht
localhost:8080/chapter-11/bokeh.html
Controls
WebGLRenderer
Scene
Bokeh

focus 17.54
aperture 0.00577
maxblur 0.029

Figure 11.15 – Bokeh focussed on the third set of cubes

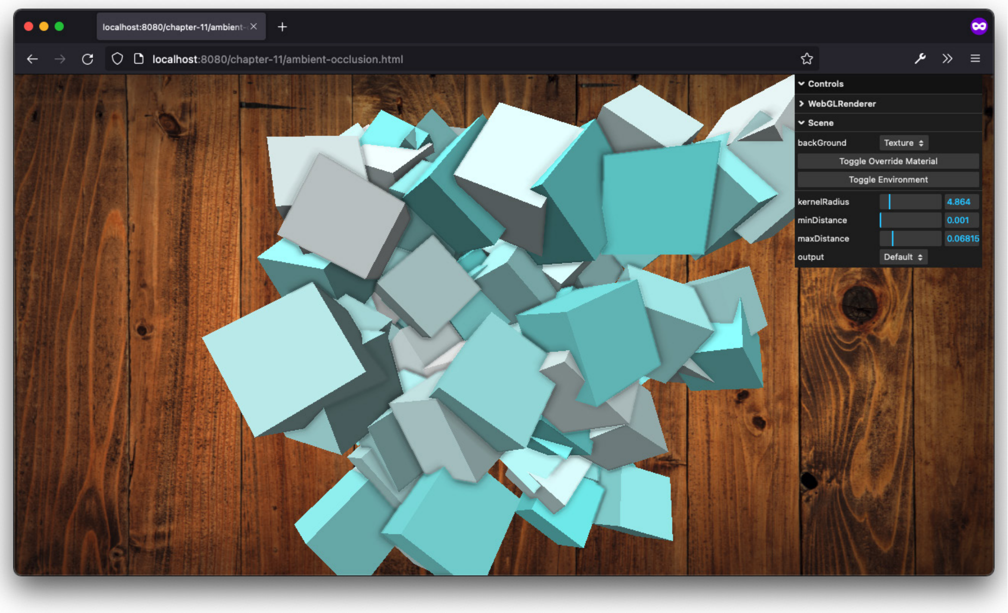
`BokehPass` can be used just like the other passes we've seen so far:

```
const params = {
   focus: 10,
   aspect: camera.aspect,
   aperture: 0.0002,
   maxblur: 1
};
const renderPass = new RenderPass(scene, camera);
const bokehPass = new BokehPass(scene, camera, params)
bokehPass.renderToScreen = true;
const composer = new EffectComposer(renderer);
composer.addPass(renderPass);
composer.addPass(bokehPass);
```

Achieving the desired result might require some fine-tuning of the properties.

Advance pass – ambient occlusion

In *Chapter 10*, *Loading and Working with Textures*, we discussed using a pre-baked **ambient occlusion map** (**aoMap**) to directly apply shadows, based on ambient lighting. Ambient occlusion involves the shadows and light intensity variations that you see on objects, since not all parts of an object receive the same amount of ambient light. Besides using `aoMap` on the material, it is also possible to use a pass on `EffectComposer` to get the same effect. If you open the `ambient-occlusion.html` example, you will see the result of using `SSAOPass`:

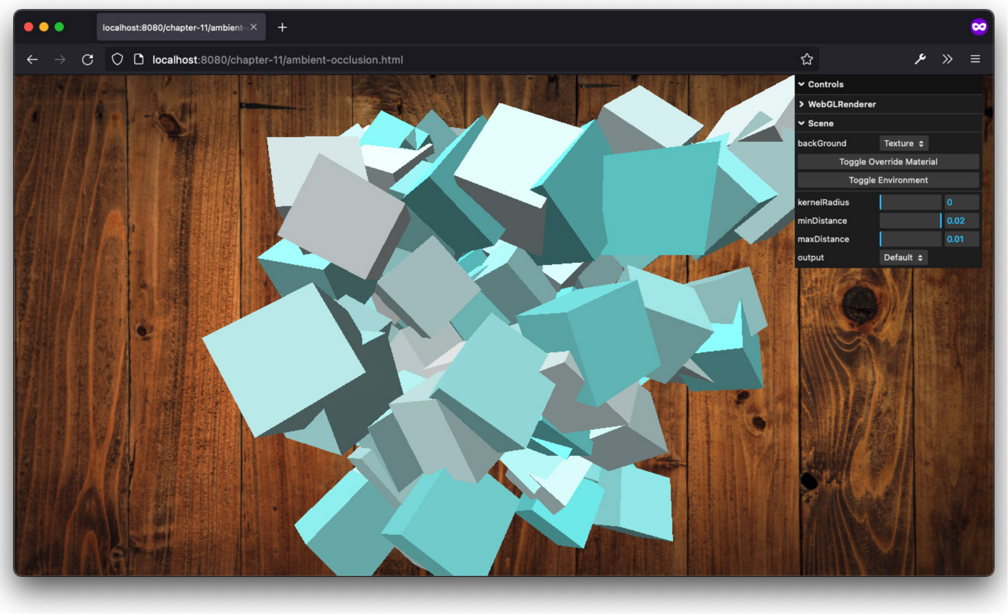

localhost:8080/chapter-11/ambient-
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment

localhost:8080/chapter-11/ambient-occlusion.html

backGround Texture
kernelRadius 4.864
minDistance 0.001
maxDistance 0.06815
output Default

Figure 11.16 – An AO pass applied

A similar scene, without the application of an ambient occlusion filter, appears to be really flat, as follows:


localhost:8080/chapter-11/ambient-
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment

localhost:8080/chapter-11/ambient-occlusion.html

backGround Texture
kernelRadius 0
minDistance 0.02
maxDistance 0.01
output Default

Figure 11.17 – The same scene without the AO pass

Note though that if you use this, you have to keep an eye on the overall performance of your application, since this is a very GPU-intensive pass.

Until now, we've used the standard passes provided by Three.js for our effects. Three.js also provides `THREE.ShaderPass`, which can be used for custom effects and comes with a large number of shaders that you can use and experiment with.

Using THREE.ShaderPass for custom effects

With `THREE.ShaderPass`, we can apply a large number of additional effects to our scene by passing in a custom shader. Three.js comes with a set of shaders that can be used together with this `THREE.ShaderPass`. They will be listed in this section. We've divided this section into three parts.

The first set involves simple shaders. All of these shaders can be viewed and configured by opening up the `shaderpass-simple.html` example:

- `BleachBypassShader`: This creates a bleach bypass effect. With this effect, a silver-like overlay will be applied to the image.
- `BlendShader`: This isn't a shader that you apply as a single postprocessing step, but it allows you to blend two textures together. You can, for instance, use this shader to smoothly blend the rendering of one scene into another (not shown in `shaderpass-simple.html`).
- `BrightnessContrastShader`: This allows you to change the brightness and contrast of an image.
- `ColorifyShader`: This applies a color overlay to the screen. We've seen this one already, in the mask example.
- `ColorCorrectionShader`: With this shader, you can change the color distribution.
- `GammaCorrectionShader`: This applies a gamma correction to the rendered scene. This uses a fixed gamma factor of 2. Note that you can also set the gamma correction directly on `THREE.WebGLRenderer` by using the `gammaFactor`, `gammaInput`, and `gammaOutput` properties.
- `HueSaturationShader`: This allows you to change the hue and saturation of the colors.
- `KaleidoShader`: This adds a kaleidoscope effect to the scene that provides radial reflection around the center of the scene.
- `LuminosityShader` and `LuminostyHighPassShader`: This provides a luminosity effect, where the luminosity of the scene is shown.
- `MirrorShader`: This creates a mirror effect for part of the screen.
- `PixelShader`: This creates a pixelated effect.
- `RGBShiftShader`: This shader separates the red, green, and blue components of a color.
- `SepiaShader`: This creates a sepia-like effect on the screen.

- `SobelOperatorShader`: This provides edge detection.
- `VignetteShader`: This applies a vignette effect. This effect shows dark borders around the center of the image.

Next, we'll look at the shaders that provide a couple of blur-related effects. These effects can be experimented with through the `shaderpass-blurs.html` example:

- `HorizontalBlurShader` and `VerticalBlurShader`: These apply a blur effect to the complete scene.
- `HorizontalTiltShiftShader` and `VerticalTiltShiftShader`: These recreate a tilt-shift effect. With the tilt-shift effect, it is possible to create scenes that look miniature by making sure that only part of the image is sharp.
- `FocusShader`: This is a simple shader that results in a sharply rendered center area with blurring along its borders.

Finally, there are a number of shaders that we won't look at in detail; we are listing them simply for the sake of completeness. These shaders are mostly used internally, by either another shader or the shader passes that we discussed at the beginning of this chapter:

- `THREE.FXAAShader`: This shader applies an anti-aliasing effect during the postprocessing phase. Use this if applying anti-aliasing during rendering is too expensive.
- `THREE.ConvolutionShader`: This shader is used internally by the `BloomPass` render pass.
- `THREE.DepthLimitedBlurShader`: This is used internally by `SAOPass` for ambient occlusion.
- `THREE.HalftoneShader`: This is used internally by `HalftonePass`.
- `THREE.SAOShader`: This provides ambient occlusion in shader form.
- `THREE.SSAOShader`: This provides an alternative approach to ambient occlusion in shader form.
- `THREE.SMAAShader`: This provides anti-aliasing to the rendered scene.
- `THREE.ToneMapShader`: This is used internally by `AdaptiveToneMappingPass`.
- `UnpackDepthRGBAShader`: This can be used to visualize encoded depth values from an RGBA texture as a visual color.

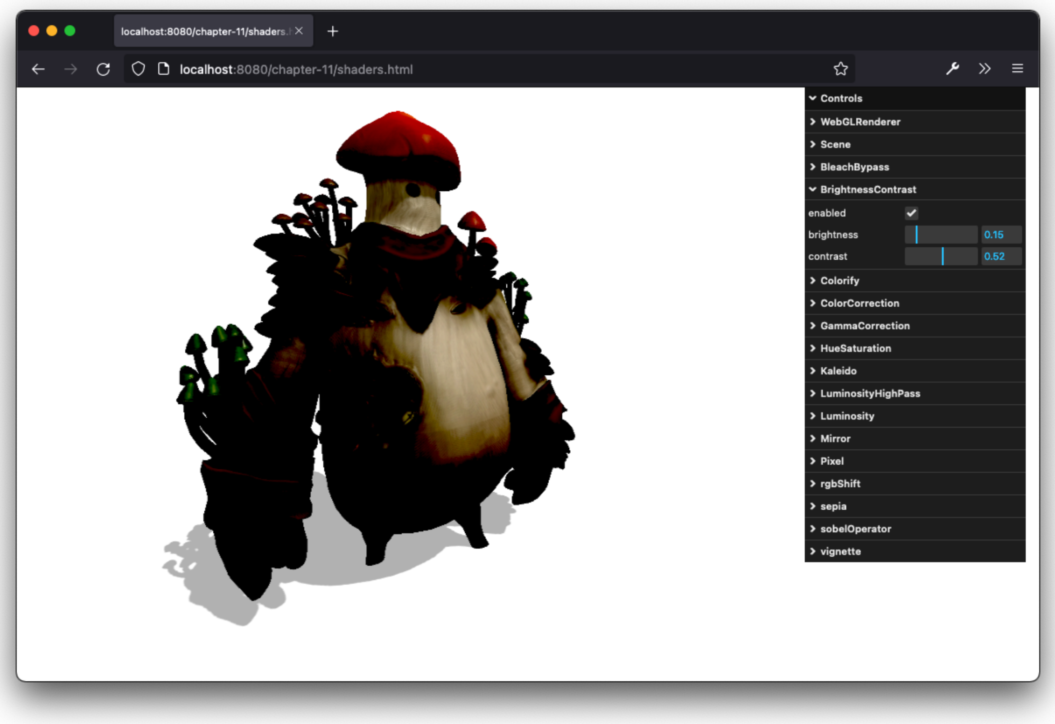
If you look through the `Shaders` directory of the Three.js distribution, you might notice a couple of other shaders that we haven't listed in this chapter. These shaders –`FresnelShader`, `OceanShader`, `ParallaxShader`, and `WaterRefractionShader` – aren't shaders that can be used for postprocessing, but they should be used with the `THREE.ShaderMaterial` object that we discussed in *Chapter 4*, *Working with Three.js Materials*.

We will start with a couple of simple shaders.

Simple shaders

To experiment with the basic shaders, we've created an example where you can play around with most of the shaders and see the effects directly in the scene. You can find this example at `shaders.html`. The following screenshots show some of the effects.

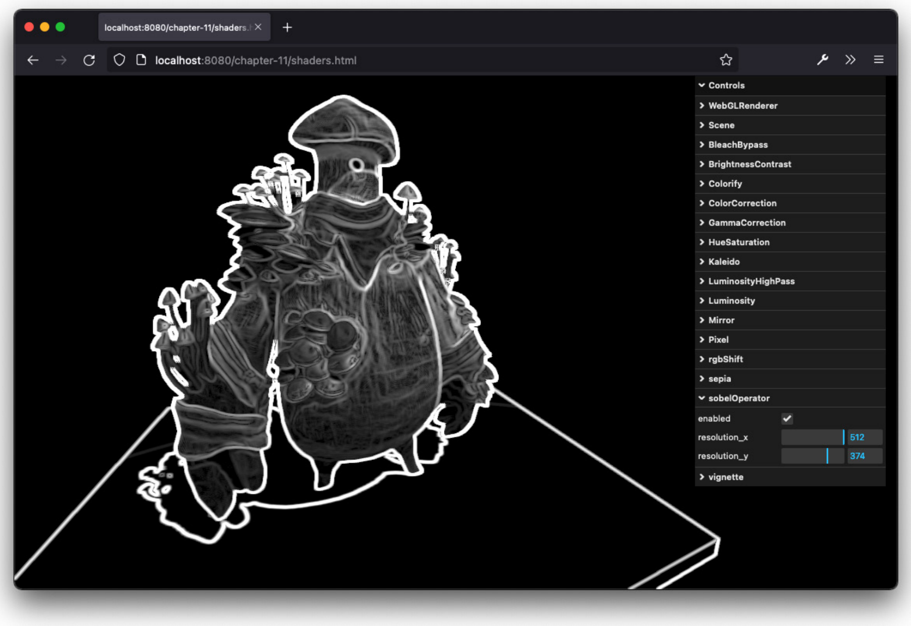
The `BrightnessContrastShader` effect is as follows:


localhost:8080/chapter-11/shaders.
localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator
vignette

enabled ☑
brightness 0.15
contrast 0.52

Figure 11.18 – The BrightnessContractShader effect

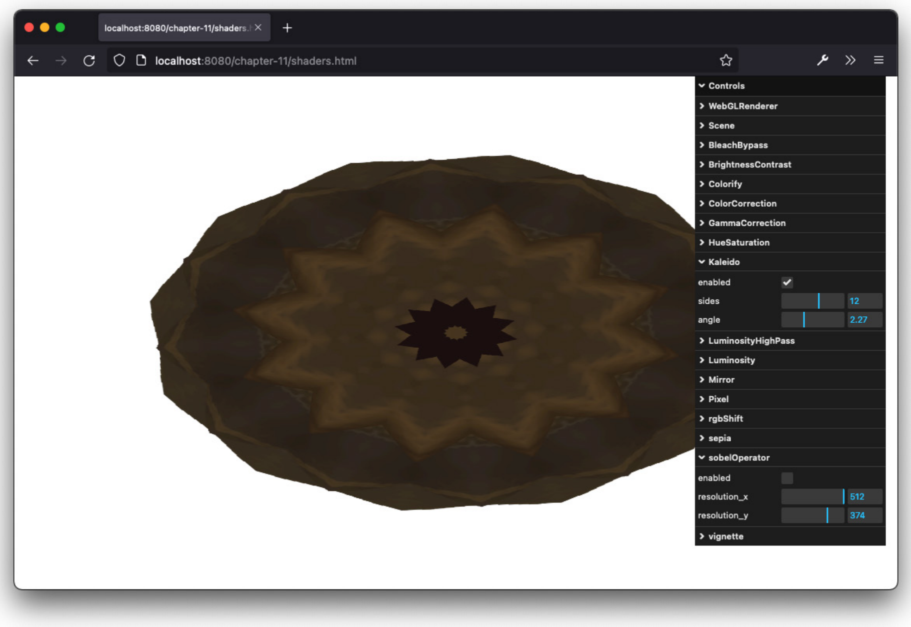
The `SobelOperatorShader` effect detects outlines:


localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator
vignette

enabled ☑
resolution_x 512
resolution_y 374

Figure 11.19 – The SobelOperatorShader effect

You can create a kaleidoscope effect using `KaleidoShader`:

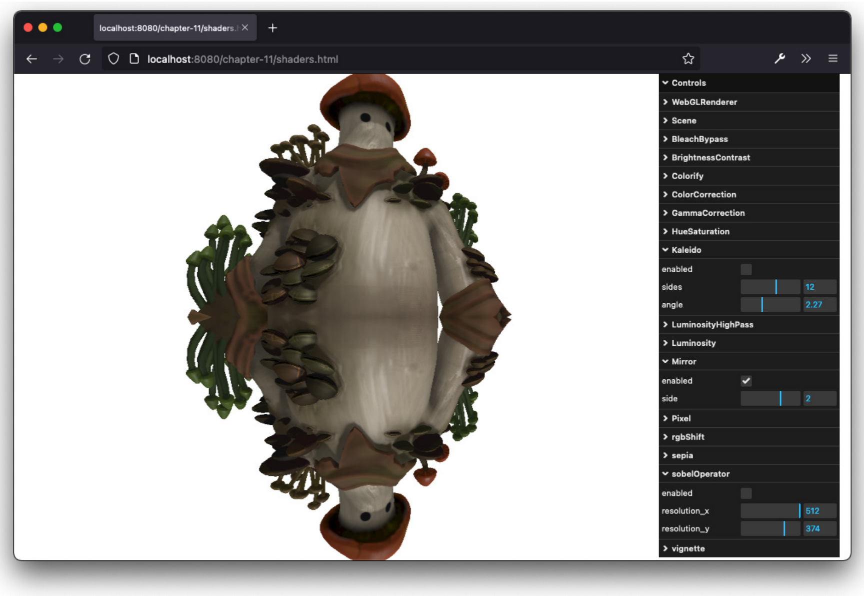

localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator
vignette

Kaleido:
enabled ☑
sides 12
angle 2.27

sobelOperator:
enabled ☐
resolution_x 512
resolution_y 374

Figure 11.20 – The KaleidoShader effect

You can mirror parts of the scene with `MirrorShader`:


localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator
vignette

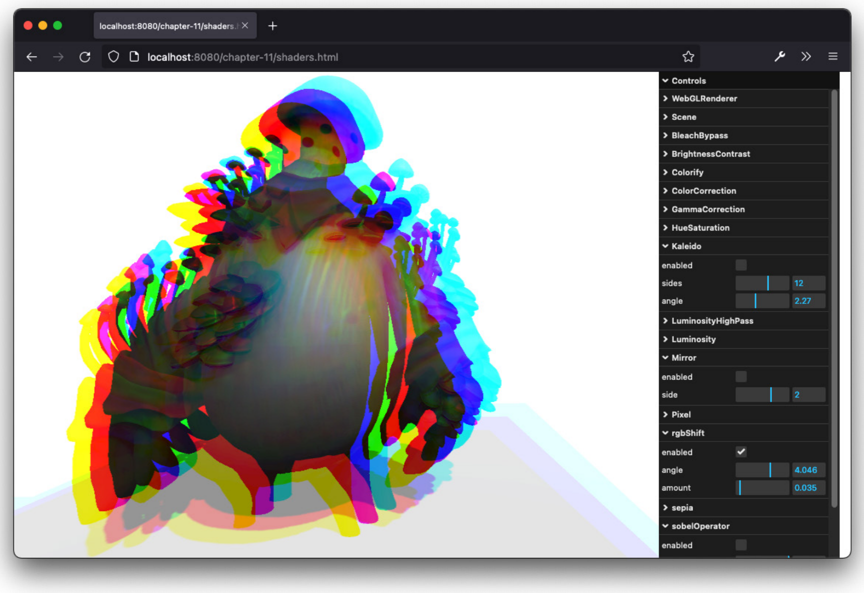
Kaleido
enabled ☐
sides 12
angle 2.27

Mirror
enabled ☑
side 2

sobelOperator
enabled ☐
resolution_x 512
resolution_y 374

Figure 11.21 – The MirrorShader effect

The `RGBShiftShader` effect looks like this:


localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator

Kaleido
enabled ☐
sides 12
angle 2.27

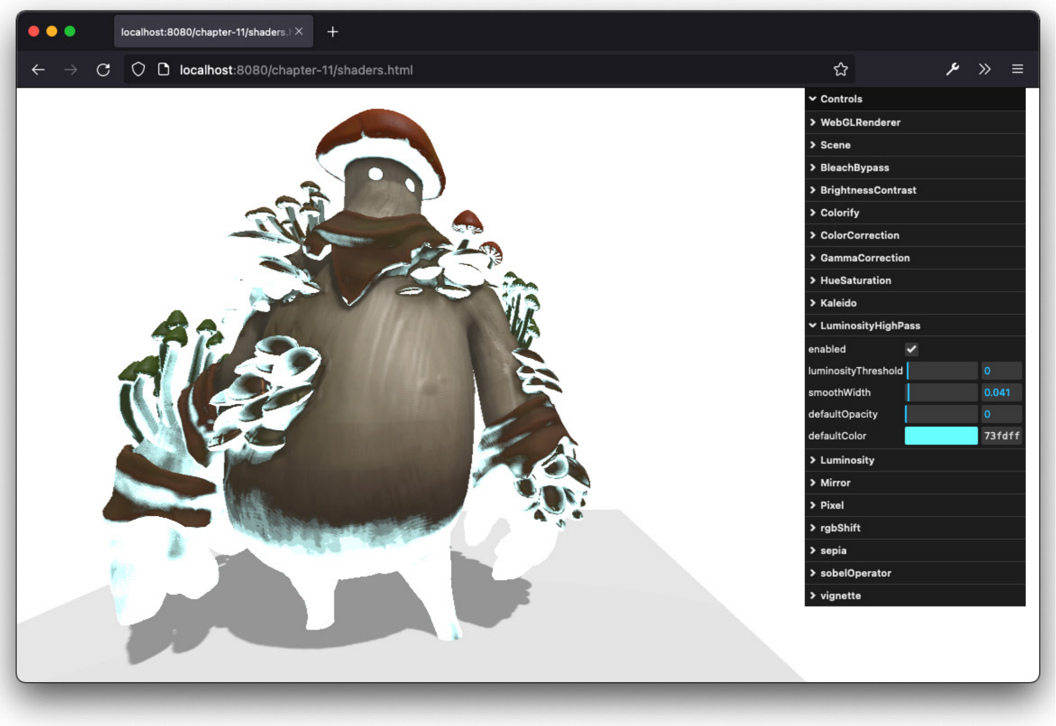
Mirror
enabled ☐
side 2

rgbShift
enabled ☑
angle 4.046
amount 0.035

sobelOperator
enabled ☐

Figure 11.22 – The RGBShiftShader effect

You can play around with the luminosity in the scene with `LuminosityHighPassShader`:


localhost:8080/chapter-11/shaders.h
localhost:8080/chapter-11/shaders.html
Controls
WebGLRenderer
Scene
BleachBypass
BrightnessContrast
Colorify
ColorCorrection
GammaCorrection
HueSaturation
Kaleido
LuminosityHighPass
Luminosity
Mirror
Pixel
rgbShift
sepia
sobelOperator
vignette

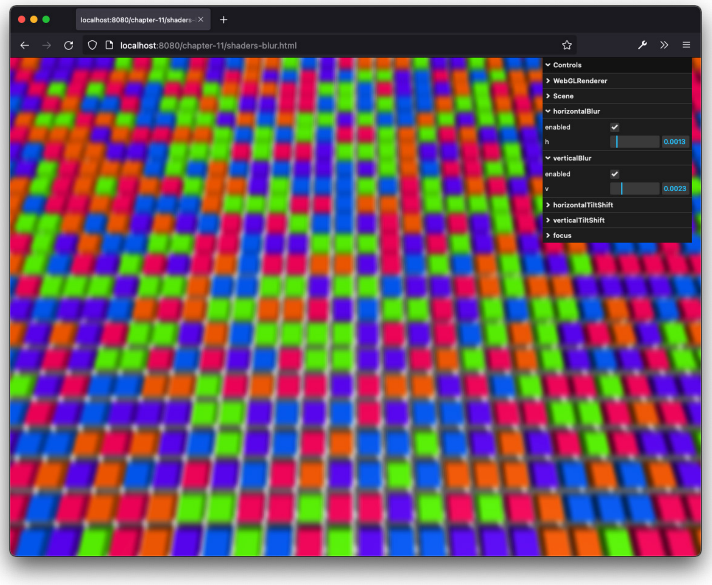
enabled ☑
luminosityThreshold 0
smoothWidth 0.041
defaultOpacity 0
defaultColor 73fdff

Figure 11.23 – The LuminosityHighPassShader effect

To see the other effects, use the menu on the right to see what they do and how they can be configured. Three.js also provides a couple of shaders that are specifically used to add blurring effects. Those are shown in the next section.

Blurring shaders

Again in this section, we won't dive into the code; we'll just show you the results of the various blur shaders. You can experiment with these by using the `shaders-blur.html` example. The first two shaders shown are `HorizontalBlurShader` and `VerticalBlurShader`:

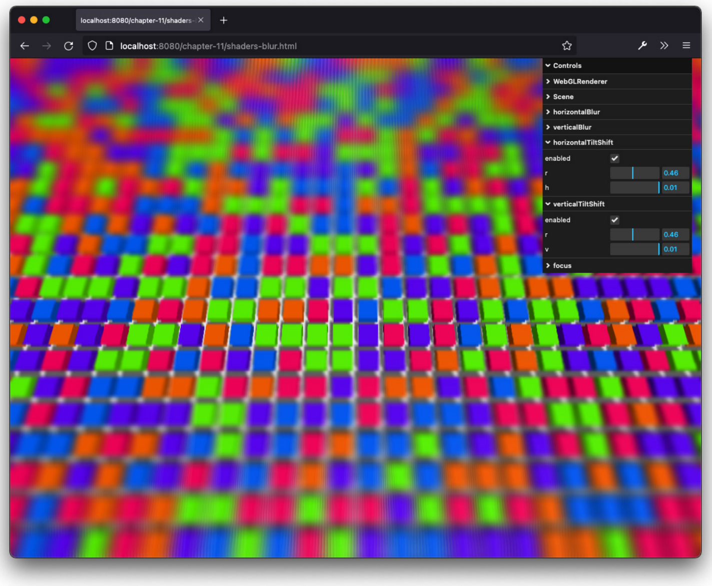

localhost:8080/chapter-11/shaders-
localhost:8080/chapter-11/shaders-blur.html
Controls
WebGLRenderer
Scene
horizontalTiltShift
verticalTiltShift
focus

horizontalBlur
enabled ☑
h 0.0013

verticalBlur
enabled ☑
v 0.0023

Figure 11.24 – HorizontalBlurShader and VerticalBlurShader

Another blur-like effect is provided by `HorizontalTiltShiftShader` and `VerticalTiltShiftShader`. This shader doesn't blur the complete scene, only a small area. This provides an effect called tilt-shift. This is often used to create miniature-like scenes from normal photographs. The following screenshot shows this effect:

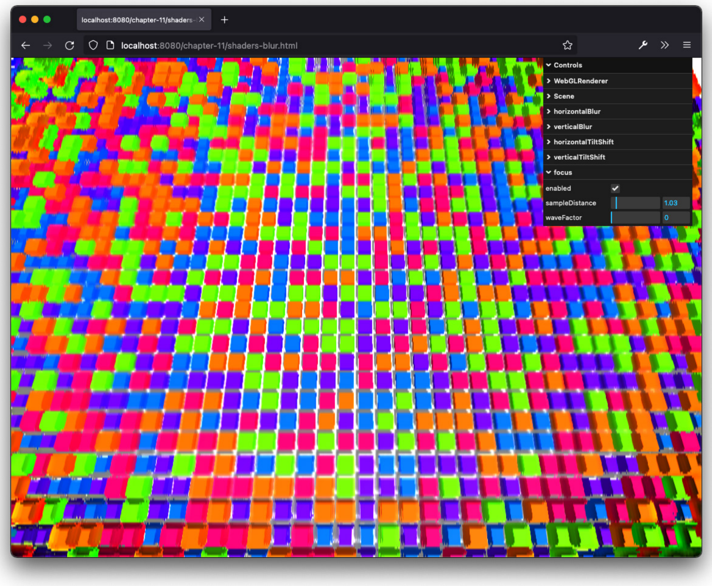

localhost:8080/chapter-11/shaders-
localhost:8080/chapter-11/shaders-blur.html
Controls
WebGLRenderer
Scene
horizontalBlur
verticalBlur
focus

horizontalTiltShift
enabled ☑
r 0.46
h 0.01

verticalTiltShift
enabled ☑
r 0.46
v 0.01

Figure 11.25 – HorizontalTiltShiftShader and VerticalTiltShiftShader

And the last of the blur-like effects is provided by `FocusShader`:


localhost:8080/chapter-11/shaders-
localhost:8080/chapter-11/shaders-blur.html
Controls
WebGLRenderer
Scene
horizontalBlur
verticalBlur
horizontalTiltShift
verticalTiltShift
focus

enabled ☑
sampleDistance 1.03
waveFactor 0

Figure 11.26 – FocusShader

So far, we've used the shaders provided by Three.js. However, it is also possible to write your own shaders for use with `THREE.EffectComposer`.

Creating custom postprocessing shaders

In this section, you'll learn how to create a custom shader that you can use in postprocessing. We'll create two different shaders. The first one will convert the current image to a grayscale image, and the second one will convert the image to an 8-bit image by reducing the number of colors that are available.

Vertex and fragment shaders

Creating vertex and fragment shaders is a very broad subject. In this section, we will only touch the surface of what can be done by these shaders and how they work. For more in-depth information, you can find the WebGL specification at `http://www.khronos.org/webgl/`. An additional resource, full of examples, is Shadertoy, available at `https://www.shadertoy.com`, or *The Book of Shaders*: `https://thebookofshaders.com/`.

Custom grayscale shader

To create a custom shader for Three.js (and also for other WebGL libraries), you have to create two components: a vertex shader and a fragment shader. The vertex shader can be used to change the position of individual vertices, and the fragment shader can be used to determine the colors of individual pixels. For a postprocessing shader, we only need to implement a fragment shader, and we can keep the default vertex shader provided by Three.js.

An important point to make before looking at the code is that GPUs support multiple shader pipelines. This means that the vertex shaders run in parallel on multiple vertices at the same time, and the same goes for the fragment shaders.

Let's start by looking at the complete source code for the shader that applies a grayscale effect to our image (`custom-shader.js`):

```
export const CustomGrayScaleShader = {
  uniforms: {
    tDiffuse: { type: 't', value: null },
    rPower: { type: 'f', value: 0.2126 },
    gPower: { type: 'f', value: 0.7152 },
    bPower: { type: 'f', value: 0.0722 }
  },
  // 0.2126 R + 0.7152 G + 0.0722 B
  // vertexshader is always the same for postprocessing
     steps
  vertexShader: [
    'varying vec2 vUv;',
    'void main() {',
    'vUv = uv;',
    'gl_Position = projectionMatrix * modelViewMatrix *
      vec4( position, 1.0 );',
    '}'
  ].join('\n'),
  fragmentShader: [
    // pass in our custom uniforms
    'uniform float rPower;',
    'uniform float gPower;',
    'uniform float bPower;',
    // pass in the image/texture we'll be modifying
    'uniform sampler2D tDiffuse;',
    // used to determine the correct texel we're working on
    'varying vec2 vUv;',
    // executed, in parallel, for each pixel
    'void main() {',
    // get the pixel from the texture we're working with
       (called a texel)
    'vec4 texel = texture2D( tDiffuse, vUv );',
    // calculate the new color
    'float gray = texel.r*rPower + texel.g*gPower +
      texel.b*bPower;',
    // return this new color
    'gl_FragColor = vec4( vec3(gray), texel.w );',
    '}'
  ].join('\n')
}
```

An alternative way of defining shaders

In *Chapter 4*, we showed how to define the shaders in separate standalone files. In Three.js, most shaders follow the structure seen in the previous code fragment. Both methods can be used to define the code of the shaders.

As you can see in the preceding code block, this isn't JavaScript. When you write shaders, you write them in the **OpenGL Shading Language** (**GLSL**), which looks a lot like the C programming language. More information on GLSL can be found at `http://www.khronos.org/opengles/sdk/docs/manglsl/`.

First, let's look at the vertex shader:

```
  vertexShader: [
    'varying vec2 vUv;',
    'void main() {',
    'vUv = uv;',
    'gl_Position = projectionMatrix * modelViewMatrix *
      vec4( position, 1.0 );',
    '}'
    ].join('\n'),
```

For postprocessing, this shader doesn't really need to do anything. The preceding code is the standard way that Three.js implements a vertex shader. It uses `projectionMatrix`, which is the projection from the camera, together with `modelViewMatrix`, which maps an object's position into the world position, in order to determine where to render a vertex on screen. For postprocessing, the only interesting thing in this piece of code is that the `uv` value, which indicates which texel to read from a texture, is passed on to the fragment shader using the `varying vec2 vUv` variable.

This can be used to get the pixel to modify in the fragment shader. Now, let's look at the fragment shader and see what the code is doing. We will start with the following variable declaration:

```
    'uniform float rPower;',
    'uniform float gPower;',
    'uniform float bPower;',
    'uniform sampler2D tDiffuse;',
    'varying vec2 vUv;',
```

Here, we can see four instances of the `uniform` property. The instances of the `uniform` property have values that are passed in from JavaScript to the shader, which are the same for each fragment that is processed. In this case, we pass in three floats, identified by type `float` (which are used to determine the ratio of a color to include in the final grayscale image), and a texture (`tDiffuse`) is also passed in, identified by type `tDiffuse`. This texture contains the image from the previous pass from the `EffectComposer` instance. Three.js makes sure this texture gets passed to this shader when `tDiffuse` is used as its name. We can also set the other instances of the `uniform` property from JavaScript by ourselves. Before we can use these uniforms from JavaScript, we have to define which `uniform` properties we want to expose to JavaScript. This is done as follows, at the top of the shader file:

```
uniforms: {
"tDiffuse": { type: "t", value: null },
"rPower":   { type: "f", value: 0.2126 },
"gPower":   { type: "f", value: 0.7152 },
"bPower":   { type: "f", value: 0.0722 }
},
```

At this point, we can receive configuration parameters from Three.js, which will provide the output of the current rendering. Let's look at the code that will convert each pixel to a gray pixel:

```
"void main() {",
"vec4 texel = texture2D( tDiffuse, vUv );",
"float gray = texel.r*rPower + texel.g*gPower +
  texel.b*bPower;", "gl_FragColor = vec4( vec3(gray),
    texel.w );"
```

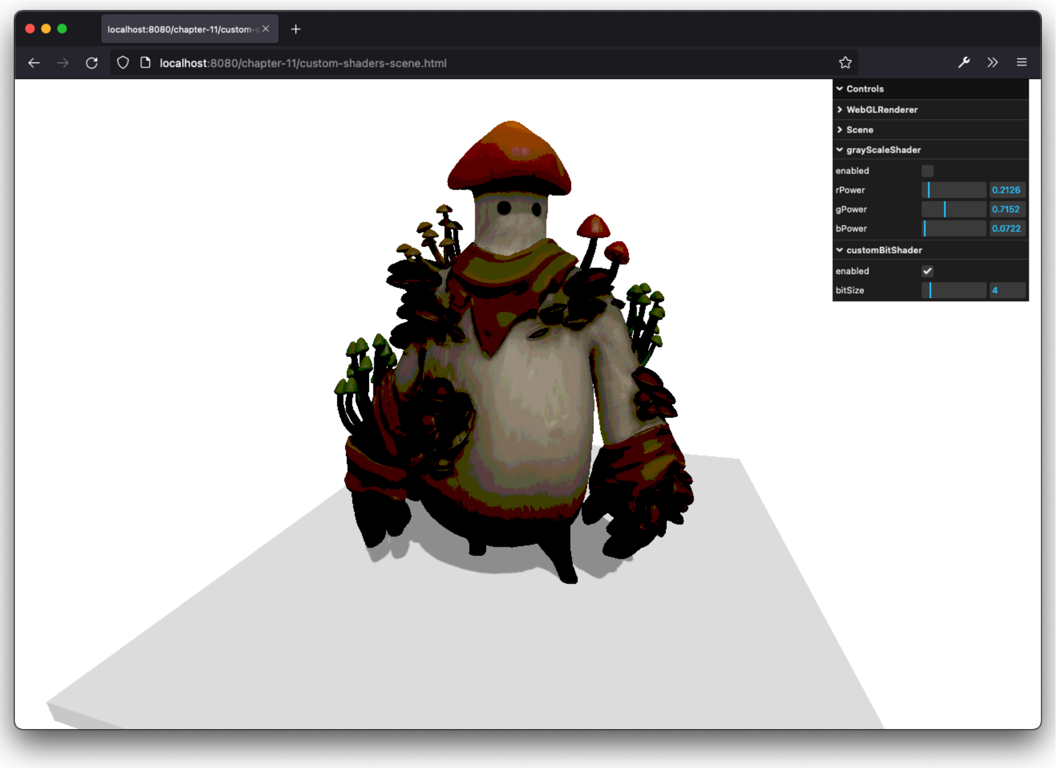
What happens here is that we get the correct pixel from the passed-in texture. We do this by using the `texture2D` function, where we pass in our current image (`tDiffuse`) and the location of the pixel (`vUv`) that we want to analyze. The result is a texel (a pixel from a texture) that contains a color and an opacity (`texel.w`). Next, we use the `r`, `g`, and `b` properties of this texel to calculate a gray value. This gray value is set to the `gl_FragColor` variable, which is eventually shown on the screen. And, with that, we have our own custom shader. This shader is used in the same way that we've already seen a couple of times in this chapter. First, we just need to set up `EffectComposer`, as follows:

```
const effectCopy = new ShaderPass(CopyShader)
effectCopy.renderToScreen = true
const grayScaleShader = new ShaderPass
  (CustomGrayScaleShader)
const gammaCorrectionShader = new ShaderPass
  (GammaCorrectionShader)
const composer = new EffectComposer(renderer)
composer.addPass(new RenderPass(scene, camera))
composer.addPass(grayScaleShader)
composer.addPass(gammaCorrectionShader)
composer.addPass(effectCopy)
```

We call `composer.render()` in the render loop. If we want to change the properties of this shader at runtime, we can just update the `uniforms` property that we've defined, as follows:

```
shaderPass.uniforms.rPower.value = ...;
shaderPass.uniforms.gPower.value = ...;
shaderPass.uniforms.bPower.value = ...;
```

The result can be seen in `custom-shaders-scene.html`. The following screenshot shows this example:


localhost:8080/chapter-11/custom-s
localhost:8080/chapter-11/custom-shaders-scene.html
Controls
WebGLRenderer
Scene

grayScaleShader
enabled ☐
rPower 0.2126
gPower 0.7152
bPower 0.0722
customBitShader
enabled ☑
bitSize 4

Figure 11.27 – A custom grayscale filter

Let's create another custom shader. This time, we'll reduce the 24-bit output to a lower bit count.

Creating a custom bit shader

Normally, colors are represented as a 24-bit value, which gives us about 16 million different colors. In the early days of computing, this wasn't possible, and the colors were often represented as 8- or 16-bit colors. With this shader, we'll automatically transform our 24-bit output to a color depth of 4 bits (or anything that you want).

Since the vertex shader is the same as in our previous example, we'll skip the vertex shader and directly list the definition of the `uniforms` property:

```
uniforms: {
  "tDiffuse": { type: "t", value: null },
  "bitSize":    { type: "i", value: 4 }
}
```

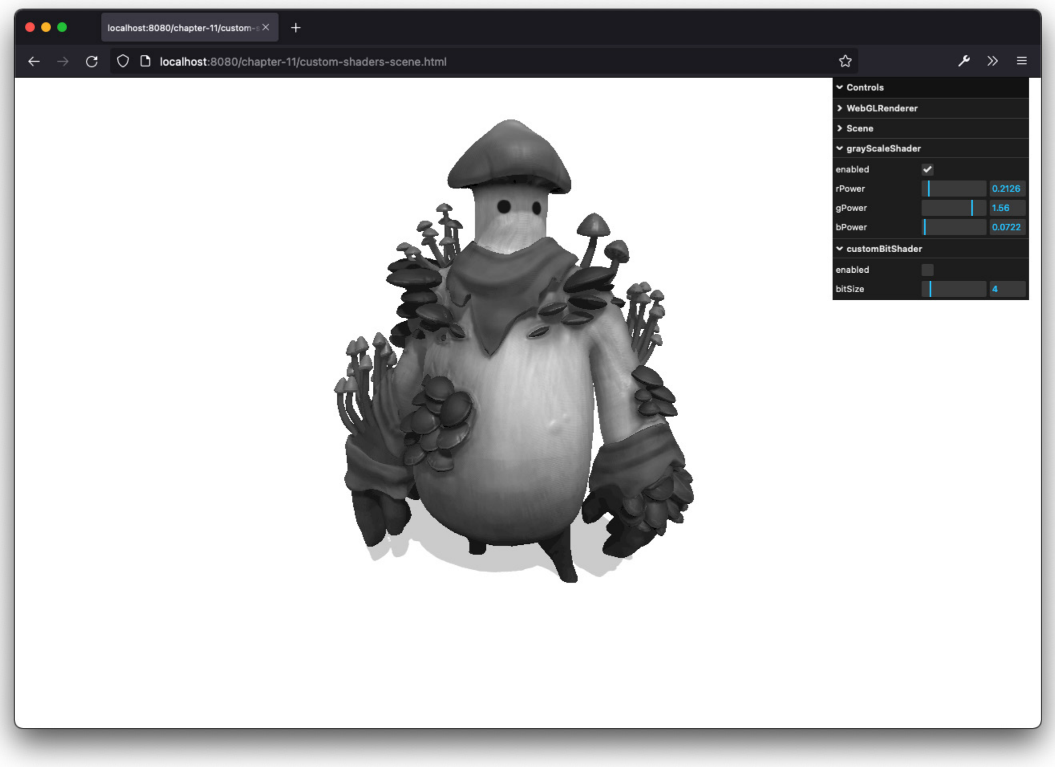
The `fragmentShader` code is as follows:

```
  fragmentShader: [
    'uniform int bitSize;',
    'uniform sampler2D tDiffuse;',
    'varying vec2 vUv;',
    'void main() {',
    'vec4 texel = texture2D( tDiffuse, vUv );',
    'float n = pow(float(bitSize),2.0);',
    'float newR = floor(texel.r*n)/n;',
    'float newG = floor(texel.g*n)/n;',
    'float newB = floor(texel.b*n)/n;',
    'gl_FragColor = vec4( vec3(newR,newG,newB), 1.0);',
    '}'
  ].join('\n')
```

We define two instances of the `uniform` property, which can be used to configure this shader. The first one is what Three.js uses to pass in the current screen, and the second one is defined by us as an integer (`type:"i"`) and serves as the color depth that we want to render the result in. The code itself is very straightforward:

1. First, we get `texel` from the `tDiffuse` texture, based on the passed-in `vUv` location of the pixel.
2. We calculate the number of colors that we can have, based on the `bitSize` property, by calculating 2 to the power of `bitSize` (`pow(float(bitSize),2.0))`).
3. Next, we calculate the new value of the color of `texel` by multiplying this value by `n`, rounding it off (`floor(texel.r*n)`), and dividing it again by `n`.
4. The result is set to `gl_FragColor` (red, green, and blue values and the opacity) and shown on the screen.

You can view the result for this custom shader in the same example as our previous custom shader, `custom-shaders-scene.html`. The following screenshot shows this example, where we set the bit size to 4. This means the model is rendered in only 16 colors:


localhost:8080/chapter-11/custom-
localhost:8080/chapter-11/custom-shaders-scene.html
Controls
WebGLRenderer
Scene
grayScaleShader
customBitShader

enabled ☑
rPower 0.2126
gPower 1.56
bPower 0.0722

enabled ☐
bitSize 4

Figure 11.28 – A custom bit filter

That's it for this chapter on postprocessing.

Summary

We discussed many different postprocessing options in this chapter. As you saw, creating `EffectComposer` and chaining passes together is actually very easy. You just have to keep a few things in mind. Not all passes will have an output on the screen. If you want to output to the screen, you can always use `ShaderPass` with `CopyShader`. The sequence in which you add passes to a composer is important. The effects are applied in that sequence. If you want to reuse the result from a specific `EffectComposer` instance, you can do so by using `TexturePass`. When you have more than one `RenderPass` in your `EffectComposer`, make sure to set the `clear` property to `false`. If not, you'll only see the output from the last `RenderPass` step. If you want to only apply an effect to a specific object, you can use `MaskPass`. When you're done with the mask, clear the mask with `ClearMaskPass`. Besides the standard passes provided by Three.js, there are also many standard shaders available. You can use these together with `ShaderPass`. Creating custom shaders for postprocessing is very easy, using the standard approach from Three.js. You only have to create a fragment shader.

We have now covered pretty much everything there is to know about the core of Three.js. In *Chapter 12*, *Adding Physics and Sounds to Your Scene*, we'll look at a library called Rapier.js, which you can use to extend Three.js with physics, in order to apply collisions, gravity, and constraints.

12

Adding Physics and Sounds to Your Scene

In this chapter, we'll look at Rapier, another library you can use to extend the basic functionality of Three.js. Rapier is a library that allows you to introduce physics into your 3D scene. By physics, we mean that your objects are subject to gravity – they can collide with one another, can be moved by applying impulses, and can be constrained in their movement by different types of joints. Besides physics, we'll also look at how Three.js can help you with adding spatial sounds to your scene.

In this chapter, we'll discuss the following topics:

- Creating a Rapier scene where your objects are subject to gravity and can collide with one another
- Showing how to change the friction and restitution (bounciness) of the objects in the scene
- Explaining the various shapes supported by Rapier and how to use them
- Showing how to create compound shapes by combining simple shapes
- Showing how a height field allows you to simulate a complex shape
- Limiting the movement of an object by using joints to connect them to other objects
- Adding sound sources to your scene, whose sound volume and direction are based on the distance to the camera

Available physics engines

There are a number of different open source JavaScript physics engines available. Most of them are not under active development though. Rapier, however, is under active development. Rapier is written in Rust and is cross-compiled to JavaScript, so you can use it in the browser. Should you choose to use any of the other libraries out there, the information in this chapter will still be useful since most of the libraries use the same approach as demonstrated in this chapter. Therefore, while the implementation and classes and functions used might be different, the concepts and setup shown in this chapter will, for the most part, be applicable regardless of the physics library you choose.

Creating a basic Three.js scene with Rapier

To get started, we created a very basic scene in which a cube drops down and hits a plane. You can see this example by looking at the `physics-setup.html` example:

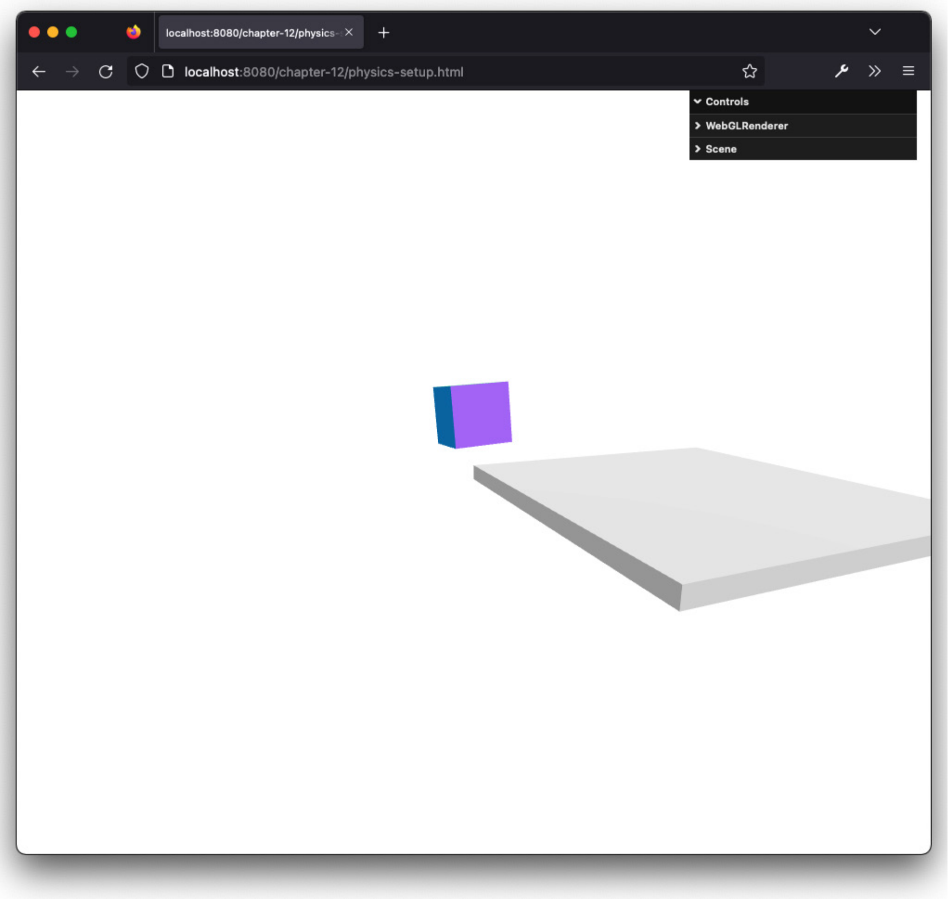

localhost:8080/chapter-12/physics-
localhost:8080/chapter-12/physics-setup.html
Controls
WebGLRenderer
Scene

Figure 12.1 – Simple Rapier physics

When you open this example, you'll see the cube slowly drop down, hit the corner of the gray horizontal plane, and bounce off it. We could have accomplished this without using a physics engine by updating the position and rotation of the cube and programming how it should react. This is, however, rather difficult to do since we need to know exactly when it hits, where it hits, and how the cube should spin away after the hit. With Rapier, we just have to configure the physical world, and Rapier will calculate exactly what happens to the objects in the scene.

Before we can configure our models to use the Rapier engine, we need to install Rapier in our project (we've already done this, so you don't have to do this if you're experimenting with the examples provided in this book):

```
$ yarn add @dimforge/rapier3d
```

Once added, we need to import Rapier into our project. This is done slightly differently than the normal imports we've seen because Rapier needs to load additional WebAssembly resources. This is needed since the Rapier library is developed in the Rust language, and compiled into WebAssembly so it can also be used on the web. To use Rapier, we need to wrap our script like this:

```
import * as THREE from 'three'
import { RigidBodyType } from '@dimforge/rapier3d'
// maybe other imports
import('@dimforge/rapier3d').then((RAPIER) => {
  // the code
}
```

This last import statement will load the Rapier library asynchronously and calls the callback when all the data has been loaded and parsed. In the rest of the code, you can just call into the `RAPIER` object to access the Rapier-specific functionality.

To set up a scene with Rapier, we need to do a few things:

1. Create a Rapier `World`. This defines the physical world that we're simulating and allows us to define the gravity that will be applied to objects in this world.
2. For each object you want to simulate with Rapier, you've got to define a `RigidBodyDesc`. This defines the position and rotation of an object in the scene (as well as some other properties). By adding this description to the `World` instance, you get back a `RigidBody`.
3. Next, you can tell Rapier the shape of the object you're adding by creating a `ColliderDesc` object. This will tell Rapier that your object is a cube, sphere, cone, or another shape; how large it is; how much friction it has with regard to other objects; and how bouncy it is. This description is then combined with the previously created `RigidBody` to create a `Collider` instance.

4. In our animation loop, we can now call `world.step()`, which makes Rapier calculate all the new positions and rotations of the `RigidBody` objects that it is aware of.

> **Online Rapier documentation**
>
> In this book, we'll look at various properties of Rapier. We won't explore the full set of features provided by Rapier since that could fill a book in itself. More information on Rapier can be found here: `https://rapier.rs/docs/`.

Let's walk through these steps and see how you combine this with the Three.js objects you're already familiar with.

Setting up the world and creating the descriptions

The first thing we need to do is create the `World` we're simulating with:

```
const gravity = { x: 0.0, y: -9.81, z: 0.0 }
const world = new RAPIER.World(gravity)
```

This is straightforward code in which we create a `World` that has a gravity of `-9.81` on the *y*-axis. This is similar to the gravity on Earth.

Next, let's define the Three.js object we saw in our example: a cube that falls and the floor that it hits:

```
const floor = new THREE.Mesh(
  new THREE.BoxGeometry(5, 0.25, 5),
  new THREE.MeshStandardMaterial({color: 0xdddddd})
)
floor.position.set(2.5, 0, 2.5)
const sampleMesh = new THREE.Mesh(
  new THREE.BoxGeometry(1, 1, 1),
  new THREE.MeshNormalMaterial()
)
sampleMesh.position.set(0, 4, 0)
scene.add(floor)
scene.add(sampleMesh)
```

Nothing new here. We just define two `THREE.Mesh` objects and position the `sampleMesh` instance, the cube, above the corner of the `floor` surface. Next, we need to create the `RigidBodyDesc` and

`ColliderDesc` objects, which represent the `THREE.Mesh` objects in the world of Rapier. We'll start with the simple one, the floor:

```
const floorBodyDesc = new RAPIER.RigidBodyDesc
  (RigidBodyType.Fixed)
const floorBody = world.createRigidBody(floorBodyDesc)
floorBody.setTranslation({ x: 2.5, y: 0, z: 2.5 })
const floorColliderDesc = RAPIER.ColliderDesc.cuboid
  (2.5, 0.125, 2.5)
world.createCollider(floorColliderDesc, floorBody)
```

Here, first, we create a `RigidBodyDesc` with a single parameter, `RigidBodyType.Fixed`. A fixed rigid body means that Rapier isn't allowed to change the position or the rotation of this object, so this object won't be affected by gravity or moved around when another object hits it. By calling `world.createRigidBody`, we add it to the `world` known by Rapier so that Rapier can take this object into account when doing its calculations. Then, we use `setTranslation` to put `RigidBody` into the same position as our Three.js floor. The `setTranslation` function takes an optional extra parameter called `wakeUp`. If `RigidBody` is sleeping (what can happen if it hasn't moved for a long time), passing in `true` for the `wakeUp` property makes sure that Rapier will take `RigidBody` into account when determining the new positions of all the objects that it is aware of.

We still need to define the shape of this object so that Rapier can tell when it collides with another object. For this, we use the `Rapier.ColliderDesc.cuboid` function in which we specify the shape. For the `cuboid` function, Rapier expects the shape to be defined by a half-width, a half-height, and a half-depth. The final step to take is to add this collider to the world and connect it to the floor. For this, we use the `world.createCollider` function.

At this point, we have defined `floor` in the Rapier world, which corresponds to the floor in our Three.js scene. Now, we define the cube that will fall in the same way:

```
Const rigidBodyDesc = new RAPIER.RigidBodyDesc
(RigidBodyType.Dynamic)
    .setTranslation(0, 4, 0)
const rigidBody = world.createRigidBody(rigidBodyDesc)
const rigidBodyColliderDesc = RAPIER.ColliderDesc.cuboid
  (0.5, 0.5, 0.5)
const rigidBodyCollider = world.createCollider
  (rigidBodyColliderDesc, rigidBody)
rigidBodyCollider.setRestitution(1)
```

This code fragment is similar to the previous one – we just create the relevant objects for Rapier that correspond to the objects in our Three.js scene. The main change here is that we used a `RigidBodyType.Dynamic` instance. This means that this object can be completely managed by Rapier. Rapier can change its position or its translation.

Additional rigid body types provided by Rapier

Besides the `Dynamic` and the `Fixed` rigid body types, Rapier also provides a `KinematicPositionBased` type, for managing the position of an object, or a `KinematicVelocityBased` type, for managing the velocity of an object ourselves. More information on this can be found here: `https://rapier.rs/docs/user_guides/javascript/rigid_bodies`.

Rendering the scene and simulating the world

What is left to do is render the Three.js object, simulate the world, and make sure that the positions of the objects managed by Rapier correspond to the position of the Three.js meshes:

```
const animate = (renderer, scene, camera) => {
  // basic animation loop
  requestAnimationFrame(() => animate(renderer, scene,
    camera))
  renderer.render(scene, camera)

  world.step()
  // copy over the position from Rapier to Three.js
  const rigidBodyPosition = rigidBody.translation()
  sampleMesh.position.set(
    rigidBodyPosition.x,
    rigidBodyPosition.y,
    rigidBodyPosition.z)
  // copy over the rotation from Rapier to Three.js
  const rigidBodyRotation = rigidBody.rotation()
  sampleMesh.rotation.setFromQuaternion(
    new THREE.Quaternion(rigidBodyRotation.x,
      rigidBodyRotation.y, rigidBodyRotation.z,
            rigidBodyRotation.w)
    )
  }
```

In our render loop, we have the normal Three.js elements to make sure we render this each step using `requestAnimationFrame`. Besides that, we call the `world.step()` function to trigger a calculation in Rapier. This will update the position and rotation of all the objects that it knows of. Next, we need to make sure that these newly calculated positions are also reflected by the Three.js objects. To do this, we just get the current position of an object in the Rapier world (`rigidBody.translation()`) and set the position of the Three.js mesh to the result of that function. For the rotation, we do the same, by calling `rotation()` on `rigidBody` first, and then applying that rotation to our Three.js mesh. Rapier works with quaternions for defining rotations, so we need to make this conversion before we can apply that rotation to the Three.js mesh.

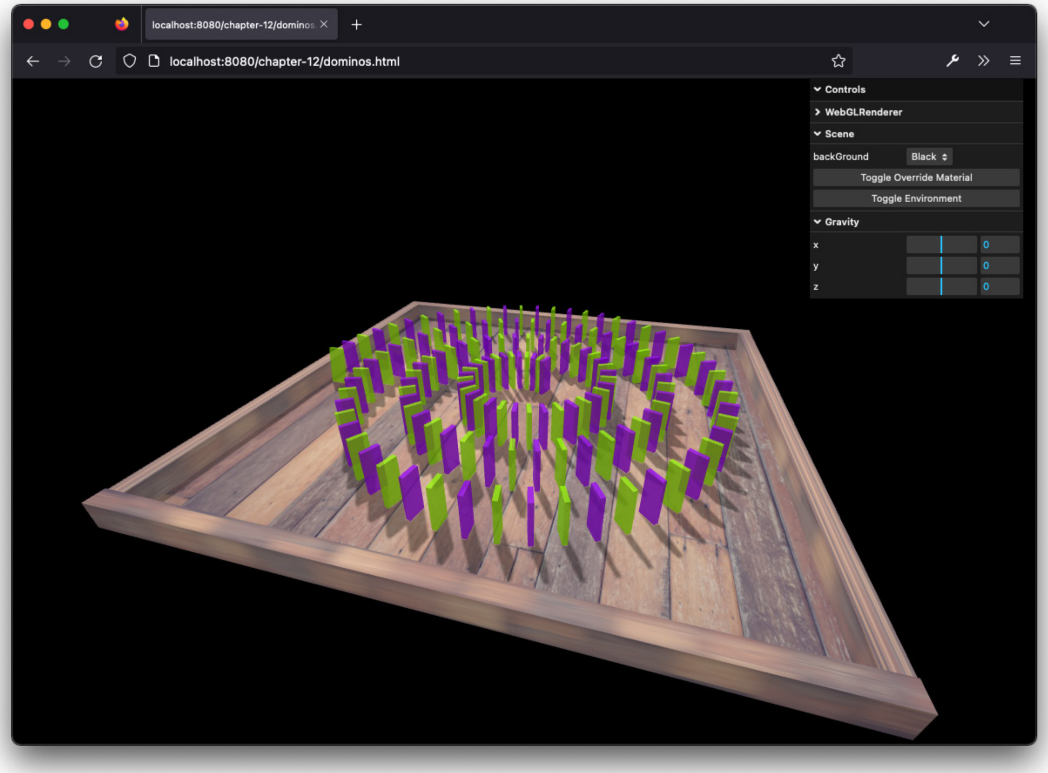
And that's all you need to do. All the examples in the following sections use this same approach:

- Setting up the Three.js scene
- Setting up a similar set of objects in the Rapier world
- Making sure that after each `step`, the location and rotation of both the Three.js scene and the Rapier world align again

In the next section, we'll expand on this example, and we'll show you more about how objects interact with each other when they collide in the Rapier world.

Simulating dominos in Rapier

The following example is built upon the same core concepts we looked at in the *Setting up the world and creating the descriptions* section. The example can be viewed by opening up the `dominos.html` example:


localhost:8080/chapter-12/dominos
localhost:8080/chapter-12/dominos.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Gravity

backGround Black

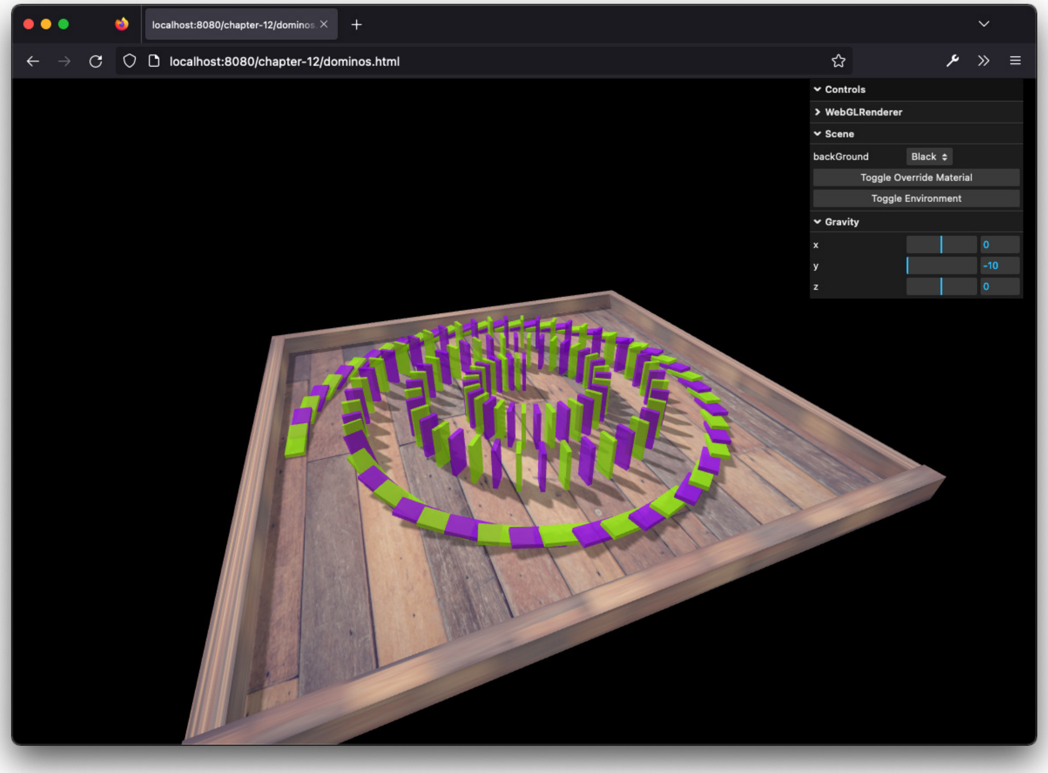
x 0

y 0

z 0

Figure 12.2 – Dominos standing still when no gravity is present

Here, you can see that we've created a simple floor on which many dominos are positioned. If you look closely, you can see that the first instance of these dominos is tilted a little bit. If we enable gravity on the *y*-axis using the menu on the right, you'll see that the first dominos falls, hits the next one, and so on until all the dominos have been knocked down:


localhost:8080/chapter-12/dominos
localhost:8080/chapter-12/dominos.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Gravity

backGround Black

x 0

y -10

z 0

Figure 12.3 – Dominos falling down after the first one is toppled

Creating this with Rapier is really straightforward. We just need to create the Three.js objects that represent the dominos, create the relevant Rapier `RigidBody` and `Collider` elements, and make sure the changes to the Rapier objects are reflected by the Three.js objects.

First, let's have a quick look at how we create the Three.js dominos:

```
const createDominos = () => {
    const getPoints = () => {
      const points = []
      const r = 2.8; const cX = 0; const cY = 0
      let circleOffset = 0
      for (let i = 0; i < 1200; i += 6 + circleOffset) {
        circleOffset = 1.5 * (i / 360)
        const x = (r / 1440) * (1440 - i) * Math.cos(i *
          (Math.PI / 180)) + cX
        const z = (r / 1440) * (1440 - i) * Math.sin(i *
          (Math.PI / 180)) + cY
        const y = 0
        points.push(new THREE.Vector3(x, y, z))
      }
      return points
    }
    const stones = new Group()
    stones.name = 'dominos'
    const points = getPoints()
    points.forEach((point, index) => {
      const colors = [0x66ff00, 0x6600ff]
      const stoneGeom = new THREE.BoxGeometry
        (0.05, 0.5, 0.2)
      const stone = new THREE.Mesh(
        stoneGeom,
        new THREE.MeshStandardMaterial({color: colors[index
        % colors.length], transparent: true, opacity: 0.8})
      )
      stone.position.copy(point)
      stone.lookAt(new THREE.Vector3(0, 0, 0))
      stones.add(stone)
    })
    return stones
  }
```

In this code fragment, we determine the position of the dominos with the `getPoints` function. This function returns a list of `THREE.Vector3` objects that represent the position of the individual stones. Each stone is placed along a spiral outward from the center. Next, these `points` are used to create a number of `THREE.BoxGeometry` objects at the same locations. To make sure the dominos are oriented correctly, we use the `lookAt` function to have them 'look at' the center of the circle. All the dominos are added to a `THREE.Group` object, which we then add to a `THREE.Scene` instance (this is not shown in the code fragment).

Now that we have our set of `THREE.Mesh` objects, we can create the corresponding set of Rapier objects:

```
const rapierDomino = (mesh) => {
  const stonePosition = mesh.position
  const stoneRotationQuaternion = new THREE.Quaternion().
    setFromEuler(mesh.rotation)
  const dominoBodyDescription = new RAPIER.RigidBodyDesc
    (RigidBodyType.Dynamic)
    .setTranslation(stonePosition.x, stonePosition.y,
      stonePosition.z)
    .setRotation(stoneRotationQuaternion))
    .setCanSleep(false)
    .setCcdEnabled(false)
  const dominoRigidBody = world.createRigidBody
    (dominoBodyDescription)
  const geometryParameters = mesh.geometry.parameters
  const dominoColliderDesc = RAPIER.ColliderDesc.cuboid(
    geometryParameters.width / 2,
    geometryParameters.height / 2,
    geometryParameters.depth / 2
  )
  const dominoCollider = world.createCollider
    (dominoColliderDesc, dominoRigidBody)
  mesh.userData.rigidBody = dominoRigidBody
  mesh.userData.collider = dominoCollider
}
```

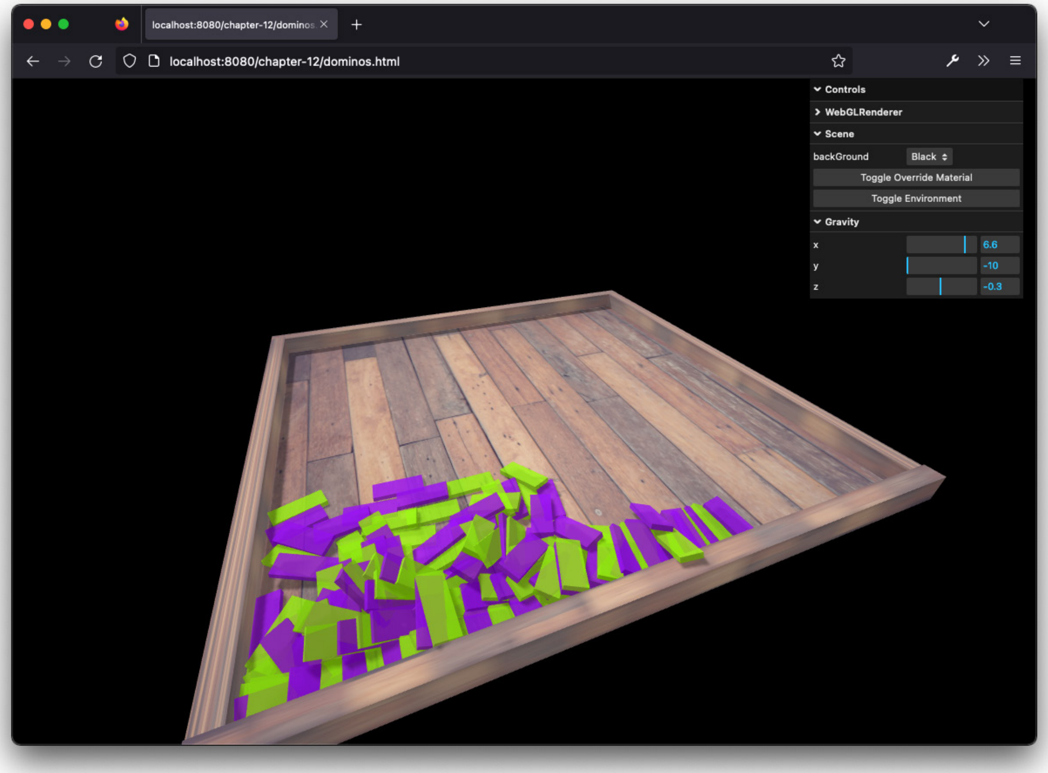
This code will look familiar to the code in *Setting up the world and creating the descriptions* section. Here, we take the position and rotation of the passed-in `THREE.Mesh` instance and use that information to create the relevant Rapier objects. To make sure we can access the `dominoCollider` and `dominoRigidBody` instances in the render loop, we add them to the `userData` property of the passed-in mesh.

The final step here is to update the `THREE.Mesh` objects in the render loop:

```
const animate = (renderer, scene, camera) => {
  requestAnimationFrame(() => animate(renderer, scene,
    camera))
  renderer.render(scene, camera)
  world.step()
  const dominosGroup = scene.getObjectByName('dominos')
  dominosGroup.children.forEach((domino) => {
    const dominoRigidBody = domino.userData.rigidBody
    const position = dominoRigidBody.translation()
    const rotation = dominoRigidBody.rotation()
    domino.position.set(position.x, position.y,
      position.z)
    domino.rotation.setFromQuaternion(new
      THREE.Quaternion(rotation.x, rotation.y,
        rotation.z, rotation.w))
  })
}
```

In each loop, we tell Rapier to calculate the next state of the world (`world.step`), and for each domino (which are `children` of the `THREE.Group` named `dominos`), we update the position and rotation of the `THREE.Mesh` instance based on the `RigidBody` object stored in that mesh's `userdata` information.

Before we move on to the most important properties provided by a collider, we'll have a quick look at how gravity affects this scene. When you open this example, with the help of the menu on the right, you can change the gravity of the world. You can use this to experiment with how the dominos respond to different gravity settings. For instance, the following example shows the situation where, after all the dominos have fallen, we increased the gravity along the *x*-axis and *z*-axis:


localhost:8080/chapter-12/dominos
localhost:8080/chapter-12/dominos.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Gravity

backGround Black

x 6.6

y -10

z -0.3

Figure 12.4 – Dominos with different gravity settings

In the next section, we'll show the effect that setting the friction and restitution has on the Rapier objects.

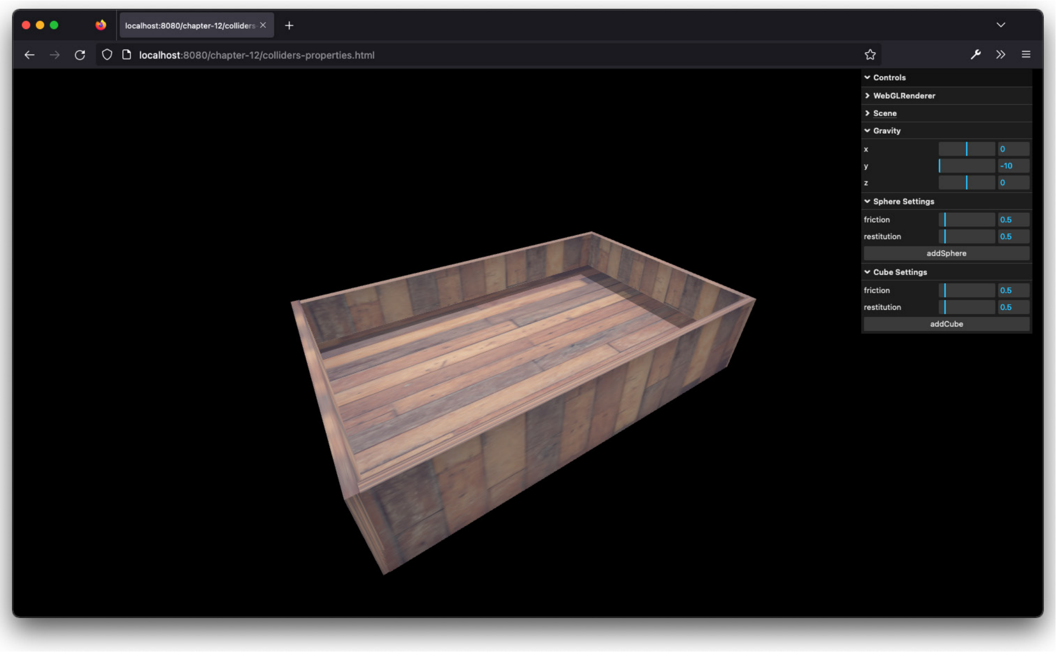
Working with restitution and friction

In the next example, we'll look a bit closer at the `restitution` and `friction` properties of the `Collider` provided by Rapier.

`restitution` is the property that defines how much energy an object keeps after it collides with another object. You can look at it a bit like bounciness. A tennis ball has high restitution, while a brick has low restitution.

`friction` defines how easily an object glides on top of another object. Objects with high friction slow down quickly when moving on top of another object, while objects with low friction can easily glide. Something such as ice has low friction, while sandpaper has high friction.

We can set these properties during the construction of the `RAPIER.ColliderDesc` object or set it afterward when we've already created the collider using the `(world.createCollider(...)` function. Before we look at the code, we'll have a look at the example. For the `colliders-properties.html` example, you'll see a big box into which you can drop shapes:


localhost:8080/chapter-12/colliders
localhost:8080/chapter-12/colliders-properties.html
Controls
WebGLRenderer
Scene
Gravity
Sphere Settings
addSphere
Cube Settings
addCube

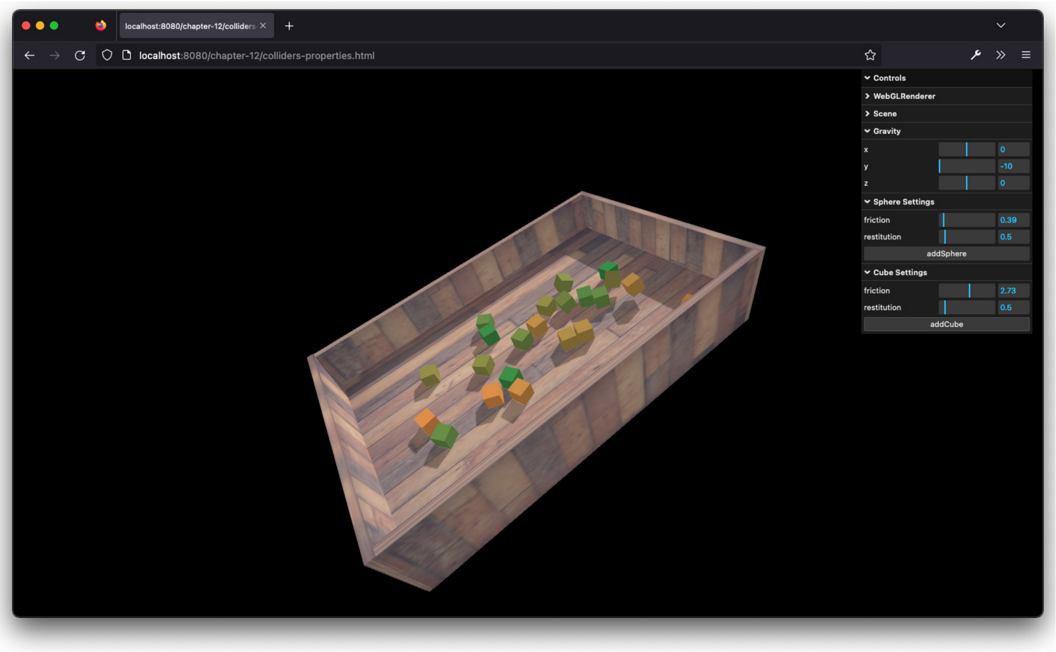
Gravity
x 0
y -10
z 0

Sphere Settings
friction 0.5
restitution 0.5

Cube Settings
friction 0.5
restitution 0.5

Figure 12.5 – Empty box to drop shapes into

With the menu on the right, you can drop in sphere and cube shapes, and set the friction and restitution for the added objects. For the first scenario, we'll add a large number of cubes with high friction.


localhost:8080/chapter-12/colliders
localhost:8080/chapter-12/colliders-properties.html
Controls
WebGLRenderer
Scene
addSphere
addCube

Gravity
x 0
y -10
z 0

Sphere Settings
friction 0.39
restitution 0.5

Cube Settings
friction 2.73
restitution 0.5

Figure 12.6 – Box with cubes with high friction

What you see here is that even though the box is moving around its axis, the cubes barely move around. This is because the cubes themselves have very high friction. If you try this with low friction, you'll see that the boxes will slide around in the bottom of the box.

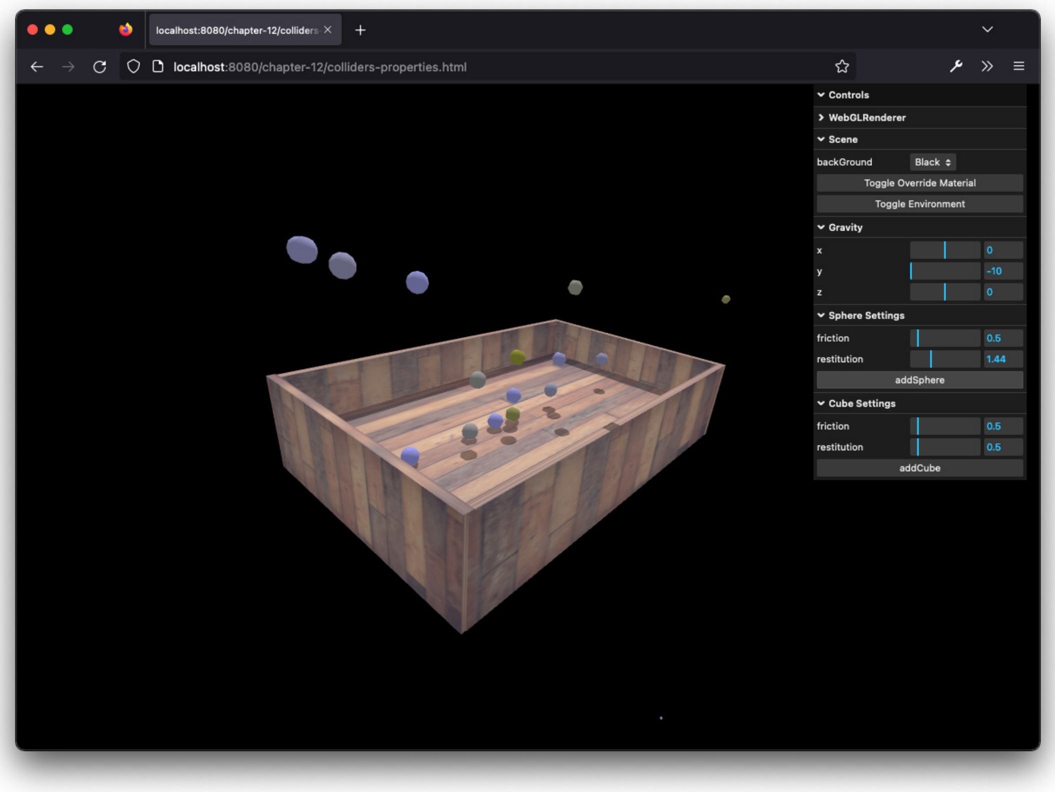
To set the friction, all you have to do is this:

```
const rigidBodyDesc = new RAPIER.RigidBodyDesc
  (RigidBodyType.Dynamic)
const rigidBody = world.createRigidBody(rigidBodyDesc)
const rigidBodyColliderDesc = RAPIER.ColliderDesc.ball(0.2)
const rigidBodyCollider = world.createCollider
  (rigidBodyColliderDesc, rigidBody)
rigidBodyCollider.setFriction(0.5)
```

Rapier provides one more way of controlling friction and that is by setting the combine rule using the `setFrictionCombineRule` function. This tells Rapier how to combine the friction of the two objects that have collided (in our example, the bottom of the box and the cube). With Rapier, you can set this to the following values:

- `CoefficientCombineRule.Average`: The average of the two coefficients is used
- `CoefficientCombineRule.Min`: The minimum among the two coefficients is used
- `CoefficientCombineRule.Multiply`: The product of the two coefficients is used
- `CoefficientCombineRule.Max`: The maximum among the two coefficients is used

To explore how `restitution` works, we can use this same example (`colliders-properties.html`):


localhost:8080/chapter-12/colliders
localhost:8080/chapter-12/colliders-properties.html
Controls
WebGLRenderer
Scene
Toggle Override Material
Toggle Environment
Gravity
Sphere Settings
addSphere
Cube Settings
addCube

backGround Black

x 0

y -10

z 0

friction 0.5

restitution 1.44

friction 0.5

restitution 0.5

Figure 12.7 – Box with spheres with high restitution

Here, we've increased the restitution of the spheres. The result is that they now bounce around in the box when added or when they hit a wall. To set the restitution, you use the same approach as for the friction:

```
const rigidBodyDesc = new RAPIER.RigidBodyDesc
  (RigidBodyType.Dynamic)
const rigidBody = world.createRigidBody(rigidBodyDesc)
const rigidBodyColliderDesc = RAPIER.ColliderDesc.ball(0.2)
const rigidBodyCollider = world.createCollider
  (rigidBodyColliderDesc, rigidBody)
rigidBodyCollider.setRestitution(0.9)
```

Rapier also allows you to set how the `restitution` property of the objects that hit each other is calculated. You can use the same values but this time, you use the `setRestitutionCombineRule` function.

The `Collider` has additional properties you can use to fine-tune how the collider interacts with the Rapier view of the world, and what happens when objects collide. Rapier itself provides very good documentation for this. Specifically for the colliders, you can find that documentation here: `https://rapier.rs/docs/user_guides/javascript/colliders#restitution`.

Rapier-supported shapes

Rapier provides a number of shapes you can use to wrap your geometries. In this section, we'll walk you through all the available Rapier shapes and demonstrate these meshes through an example. Note that to use these shapes you need to call `RAPIER.ColliderDesc.roundCuboid`, `RAPIER.ColliderDesc.ball`, and so on.

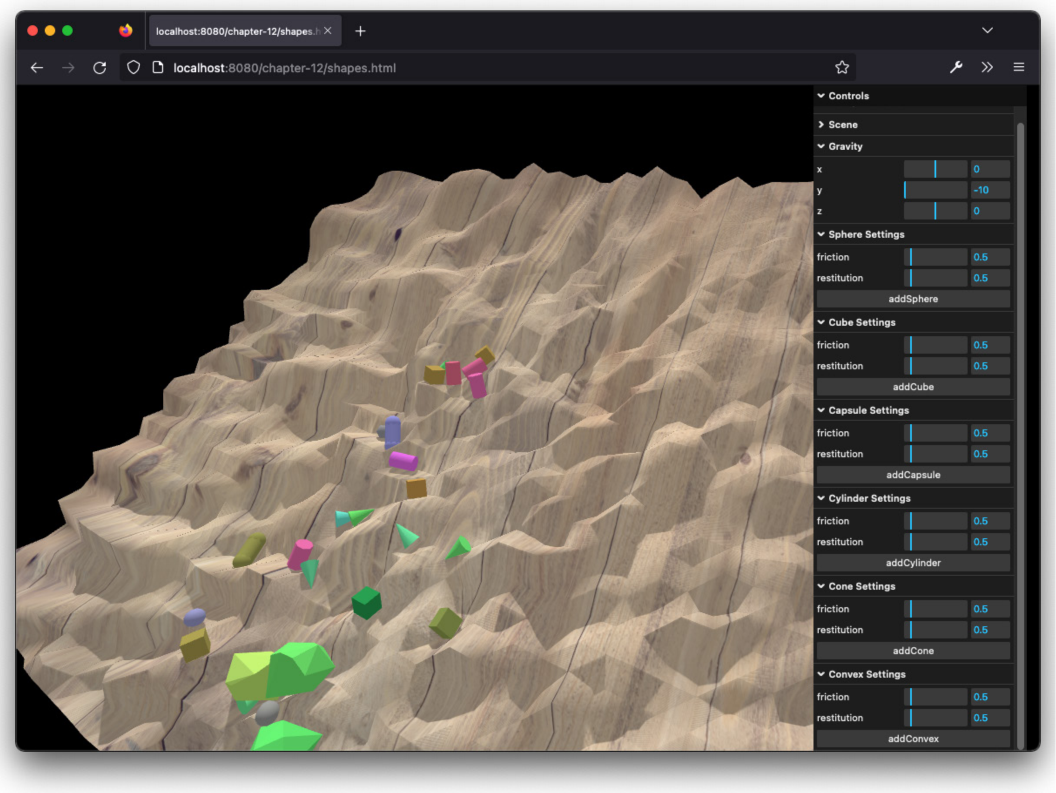
Rapier provides 3D shapes and 2D shapes. We'll only look at the 3D shapes provided by Rapier:

- `ball`: A ball shape, configured by setting the radius of the ball
- `capsule`: A capsule shape, defined by the half-height of the capsule and its radius
- `cuboid`: A simple cube shape defined by passing in the half-width, half-height, and half-depth of the shape
- `heightfield`: A height field is a shape for which each provided value defines the height of a 3D plane
- `cylinder`: A cylinder shape defined by the half-height and the radius of the cylinder
- `cone`: A cone shape defined by the half-height and the radius of the bottom of the cylinder

- `convexHull`: A convex hull is the smallest shape that encompasses all the passed-in points
- `convexMesh`: A convex mesh also takes a number of points, but it is assumed that these points already form a convex hull, so Rapier won't make any calculations to determine the smaller shape

Besides these shapes, Rapier also provides an additional rounded variant for a few of these shapes: `roundCuboid`, `roundCylinder`, `roundCone`, `roundConvexHull`, and `roundConvexMesh`.

We've provided another example in which you can what these shapes look like and how they interact when they collide with each other. Open up the `shapes.html` example to see this in action:


localhost:8080/chapter-12/shapes.h
localhost:8080/chapter-12/shapes.html
Controls
Scene
Gravity
Sphere Settings
addSphere
Cube Settings
addCube
Capsule Settings
addCapsule
Cylinder Settings
addCylinder
Cone Settings
addCone
Convex Settings
addConvex

Gravity
x 0
y -10
z 0

Sphere Settings
friction 0.5
restitution 0.5

Cube Settings
friction 0.5
restitution 0.5

Capsule Settings
friction 0.5
restitution 0.5

Cylinder Settings
friction 0.5
restitution 0.5

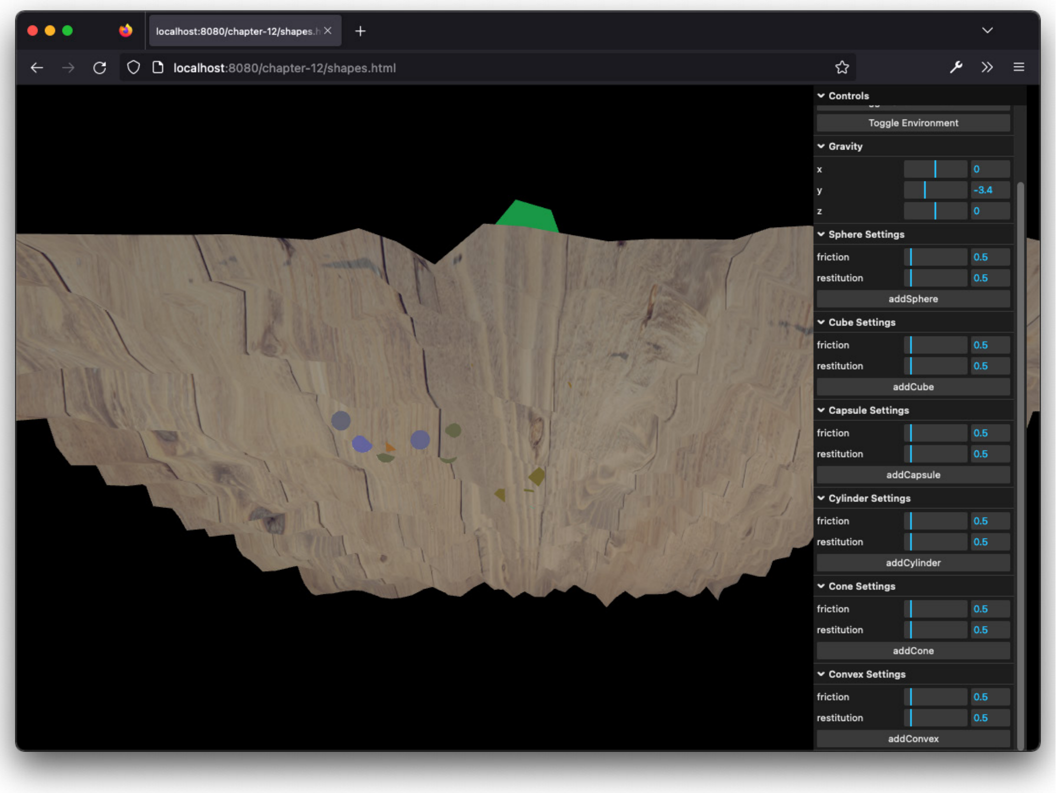
Cone Settings
friction 0.5
restitution 0.5

Convex Settings
friction 0.5
restitution 0.5

Figure 12.8 – Shapes on top of a heightfield object

When you open this example, you'll see an empty `heightfield` object. With the menu on the right, you can add different shapes and they'll collide with each other and with the `heightfield` instance. Once again, you can set the specific `restitution` and `friction` values for the objects you're adding. Since we've already explained in the previous sections how to add the shape in Rapier and make sure the corresponding shapes in Three.js are updated, we won't go into detail here on how to create the shapes from the previous list. For the code, look at the `shapes.js` file in the sources for this chapter.

One final note before we move on to the section on joints – when we want to depict simple shapes (for example, balls or cubes), the way that Rapier defines this model and the way Three.js defines it are pretty much the same. Therefore, when this kind of object collides with another object, it will look correct. When we have more complex shapes, as, in this example, with a `heightmap` instance, there can be slight differences in how Three.js interprets and interpolates these points to a `heightmap` instance, and how Rapier does so. You can see this for yourself by looking at the `shapes.html` example, adding a lot of different shapes, and then looking at the underside of the `heightfield`:


localhost:8080/chapter-12/shapes.h
localhost:8080/chapter-12/shapes.html
Controls
Toggle Environment
addSphere
addCube
addCapsule
addCylinder
addCone
addConvex

Gravity
x 0
y -3.4
z 0

Sphere Settings
friction 0.5
restitution 0.5

Cube Settings
friction 0.5
restitution 0.5

Capsule Settings
friction 0.5
restitution 0.5

Cylinder Settings
friction 0.5
restitution 0.5

Cone Settings
friction 0.5
restitution 0.5

Convex Settings
friction 0.5
restitution 0.5

Figure 12.9 – Bottom of the heightfield

What you can see here is that we can see small parts of the different objects poking through the `heightmap`. The reason is that Rapier has a different way of determining the exact shape of the `heightmap` than Three.js. In other words, Rapier thinks that the `heightmap` looks slightly different than Three.js does. Therefore, when it determines where specific shapes are when they collide, it can result in small details such as this. However, by tuning the sizes or creating simpler objects, this can easily be circumvented.

So far, we've looked at gravity and collisions. Rapier also provides a way to limit the movement and rotation of rigid bodies. We'll explain how Rapier does this by using joints.

Using joints to limit the movement of objects

Up until now, we've seen some basic physics in action. We've seen how various shapes respond to gravity, friction, and restitution, and how this affects collision. Rapier also provides advanced constructs that allow you to limit the movement of your objects. In Rapier, these objects are called joints. The following list gives an overview of the joints that are available in Rapier:

- **Fixed joint**: A fixed joint makes sure that two bodies don't move relative to one another. This means that the distance and rotation between these two objects will always be the same.
- **Spherical joint**: A spherical joint makes sure that the distance between two bodies stays the same. The bodies, however, can move around one another on all three axes.
- **Revolute joint**: With this joint, the distance between the two bodies stays the same, and they are allowed to rotate on a single axis – for instance, a steering wheel, which can only rotate around a single axis.
- **Prismatic joint**: Similar to the revolute joint but this time, the rotations between the objects are fixed, and the objects can move on a single axis. This causes a sliding effect – for example, such as a lift moving upward.

In the following sections, we'll explore these joints and see them in action in the examples.

Connecting two objects with a fixed joint

The simplest of joints is a fixed joint. With this joint, you can connect two objects, and they'll stay at the same distance and orientation that is specified when this joint is created.

This is shown in the `fixed-joint.html` example:

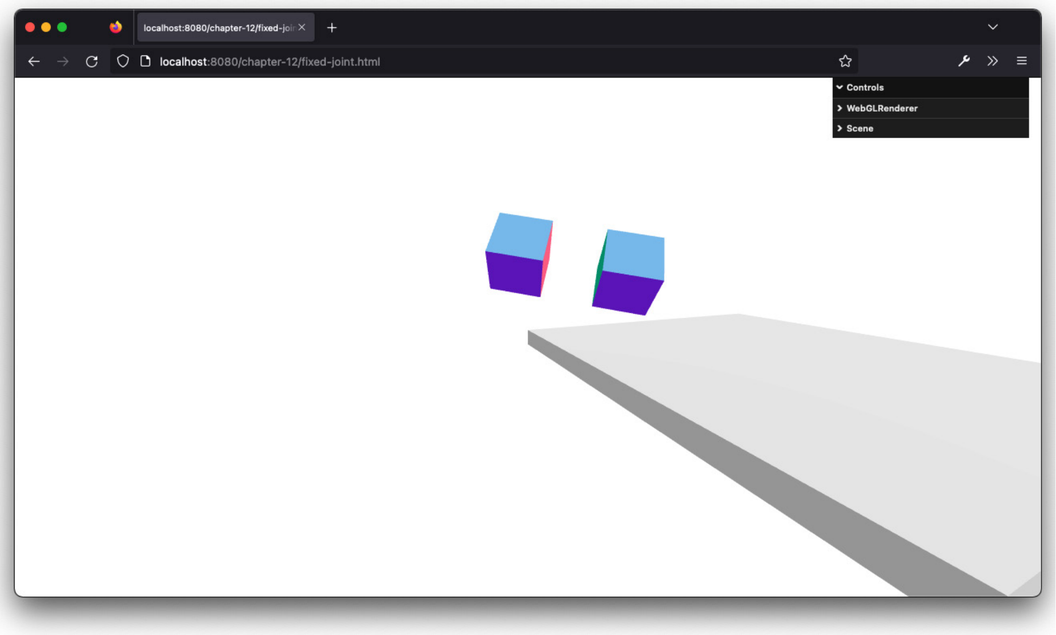

localhost:8080/chapter-12/fixed-joi
localhost:8080/chapter-12/fixed-joint.html
Controls
WebGLRenderer
Scene

Figure 12.10 – A fixed joint connecting two joints

As you can see in this example, the two cubes move as one. This happens because they are connected by a fixed joint. To set this up, we first have to create the two `RigidBody` objects and the two `Collider` objects, as we've already seen in the previous sections. The next thing we need to do is connect these two objects. For this, we first need to define `JointData`:

```
let params = RAPIER.JointData.fixed(
  { x: 0.0, y: 0.0, z: 0.0 },
  { w: 1.0, x: 0.0, y: 0.0, z: 0.0 },
  { x: 2.0, y: 0.0, z: 0.0 },
  { w: 1.0, x: 0.0, y: 0.0, z: 0.0 }
)
```

This means that we connect the first object at the position of `{ x: 0.0, y: 0.0, z: 0.0 }` (its center) to the second object, which is positioned at `{ x: 2.0, y: 0.0, z: 0.0 }`, where the first object is rotated with a quaternion of `{ w: 1.0, x: 0.0, y: 0.0, z: 0.0 }` and the second object is rotated the same amount – `{ w: 1.0, x: 0.0, y: 0.0, z: 0.0 }`. The only thing that we need to do now is tell the Rapier `world` about this joint and which `RigidBody` objects it applies to:

```
world.createImpulseJoint(params, rigidBody1, rigidBody2,
  true)
```

The last property here defines whether the `RigidBody` should wake up because of this joint. A `RigidBody` can be put to sleep when it hasn't moved for a couple of seconds. For joints, it is usually best to just set this to `true` since this makes sure that if one of the `RigidBody` objects to which we attach the joint is sleeping, `RigidBody` will wake up.

Another great way of seeing this joint in action is by using the following parameters:

```
  let params = RAPIER.JointData.fixed(
    { x: 0.0, y: 0.0, z: 0.0 },
    { w: 1.0, x: 0.0, y: 0.0, z: 0.0 },
    { x: 2.0, y: 2.0, z: 2.0 },
    { w: 0.3, x: 1, y: 1, z: 1 }
  )
```

This will cause the two cubes to get caught on the floor in the center of the scene:

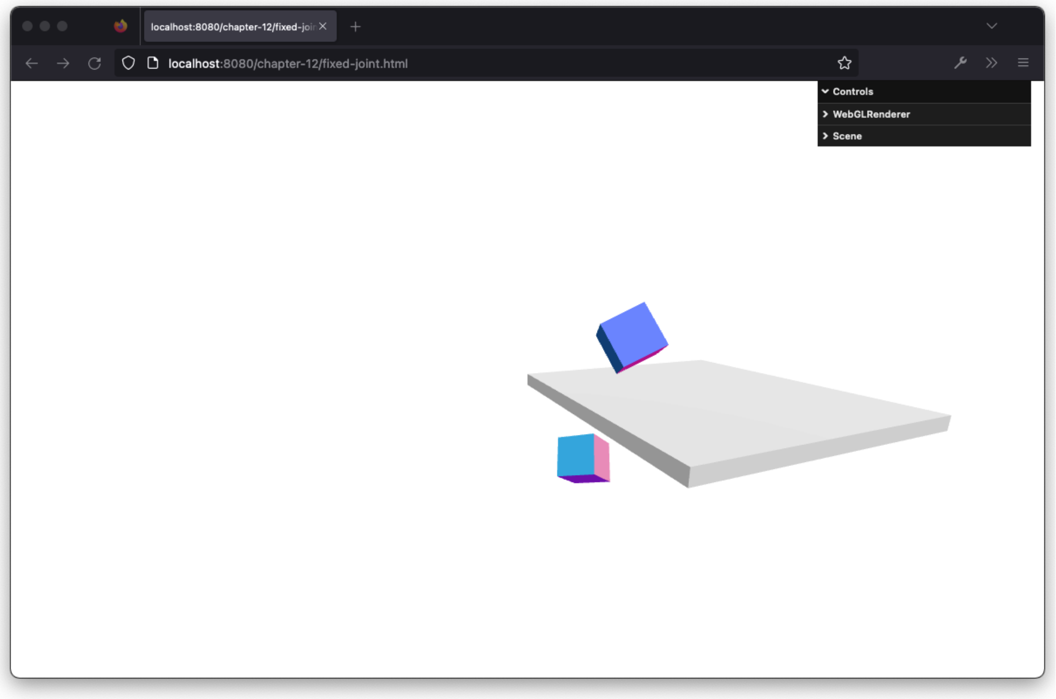
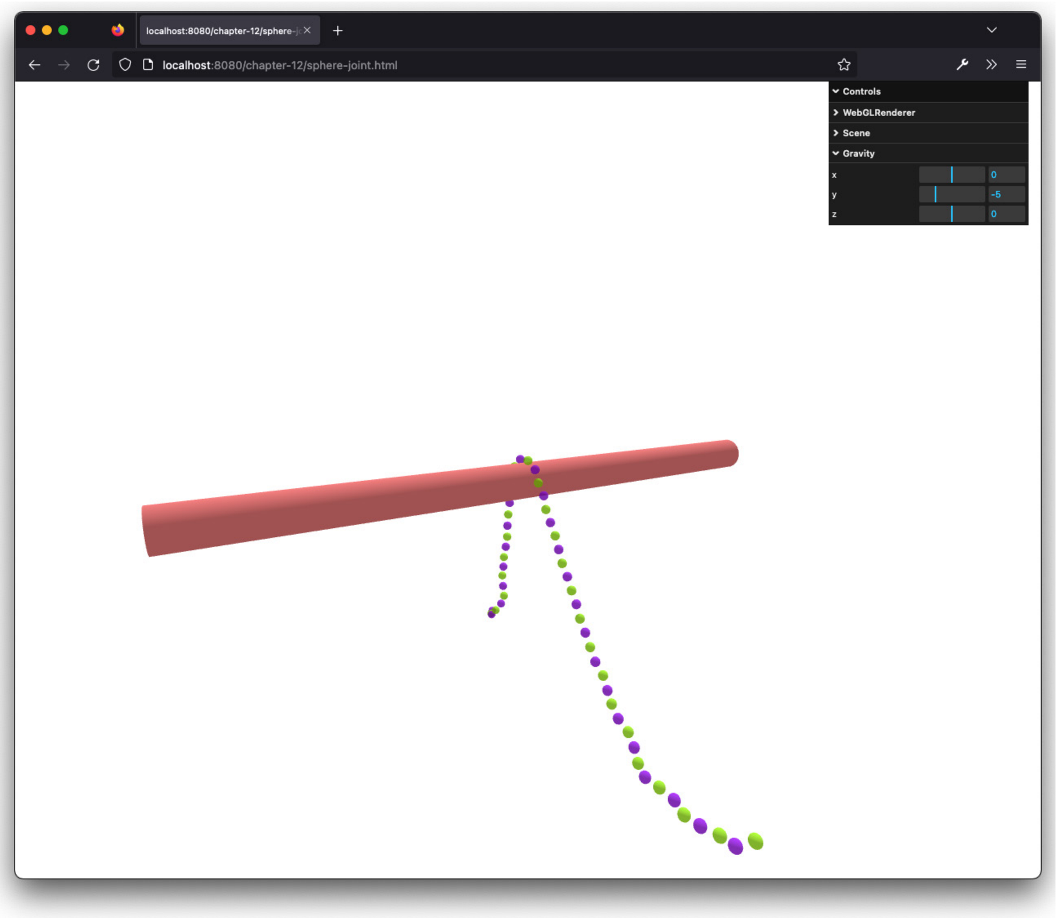

localhost:8080/chapter-12/fixed-joi
localhost:8080/chapter-12/fixed-joint.html
Controls
WebGLRenderer
Scene

Figure 12.11 – A fixed joint connecting two cubes

Next on our list is the spherical joint.

Connecting objects with a spherical joint

A spherical joint allows two objects to freely move around one another while keeping the same distance between these objects. This can be used for ragdoll effects or, as we did in this example, creating a chain (`sphere-joint.html`):


localhost:8080/chapter-12/sphere-j
localhost:8080/chapter-12/sphere-joint.html
Controls
WebGLRenderer
Scene
Gravity

x 0
y -5
z 0

Figure 12.12 – Multiple spheres connected by a spherical joint

As you can see in this example, we've connected a large number of spheres to create a chain of spheres. When these spheres hit the cylinder in the middle, they'll wrap around and slowly glide off this cylinder. You can see that while the orientation between these spheres changes based on their collisions, the absolute distance between the spheres stays the same. So, to set up this example, we've created a number of spheres with `RigidBody` and `Collider`, similar to the previous examples. For each set of two spheres, we also create a joint like this:

```
const createChain = (beads) => {
  for (let i = 1; i < beads.length; i++) {
    const previousBead = beads[i - 1].userData.rigidBody
    const thisBead = beads[i].userData.rigidBody
    const positionPrevious = beads[i - 1].position
```

```
      const positionNext = beads[i].position
      const xOffset = Math.abs(positionNext.x -
        positionPrevious.x)
      const params = RAPIER.JointData.spherical(
        { x: 0, y: 0, z: 0 },
        { x: xOffset, y: 0, z: 0 }
        )
      world.createImpulseJoint(params, thisBead,
        previousBead, true)
    }
  }
```

You can see that we create a joint using `RAPIER.JointData.spherical`. The parameters here define the position of the first object, `{ x: 0, y: 0, z: 0 }`, and the relative position of the second object, `{ x: xOffset, y: 0, z: 0 }`. We do this for all the objects and add the joints to the rapier world using `world.createImpulseJoint(params, thisBead, previousBead, true)`.

The result is that we've got a chain of spheres that is connected using these spherical joints.

The next joint, the revolute joint, allows us to restrict the movement of two objects by specifying a single axis around which an object is allowed to rotate in relation to another object.

Limiting rotation with a revolute joint

With a revolute joint, it is easy to create gear, wheel, and fan-like constructs that rotate around a single axis. The easiest way to explain this is by looking at the `revolute-joint.html` example:

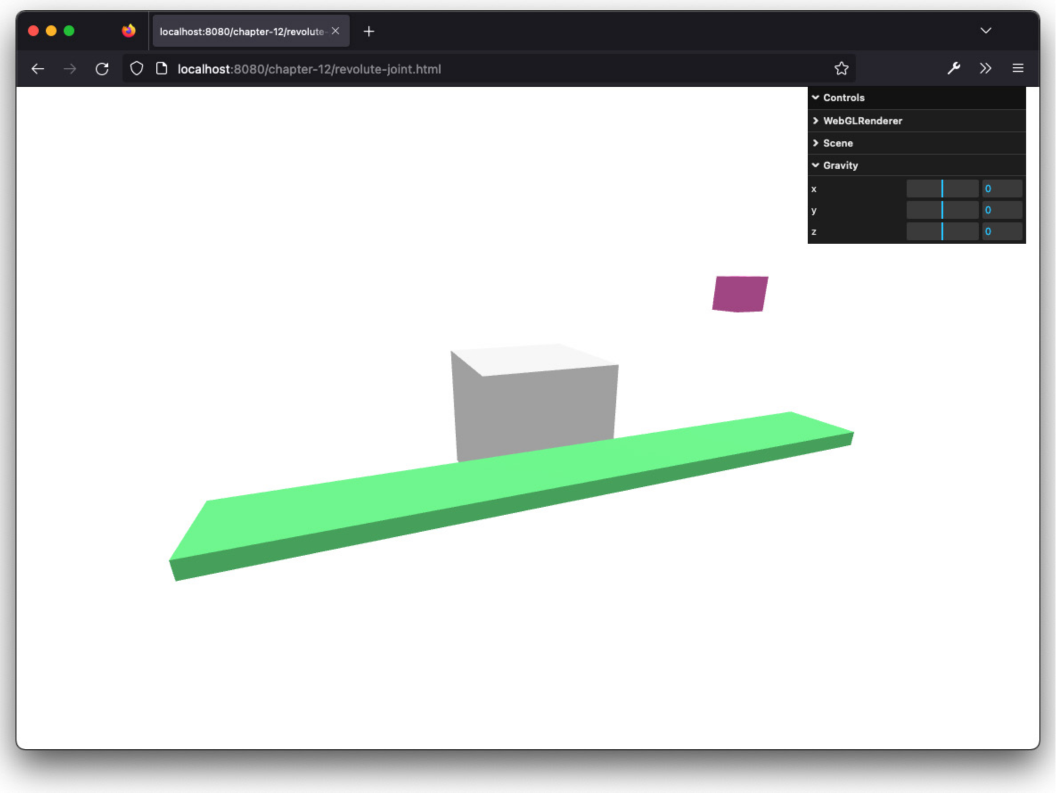

localhost:8080/chapter-12/revolute
localhost:8080/chapter-12/revolute-joint.html
Controls
WebGLRenderer
Scene
Gravity
x
0
y
0
z
0

Figure 12.13 – A cube before it is dropped on a rotating bar

In *Figure 12.13*, you can see a purple cube hovering above a green bar. When you enable gravity in the y direction, the cube will drop on top of the green bar. The center of this green bar is connected to the fixed cube in the middle using a revolute joint. The result is that this green bar will now slowly rotate because of the weight of the purple cube:

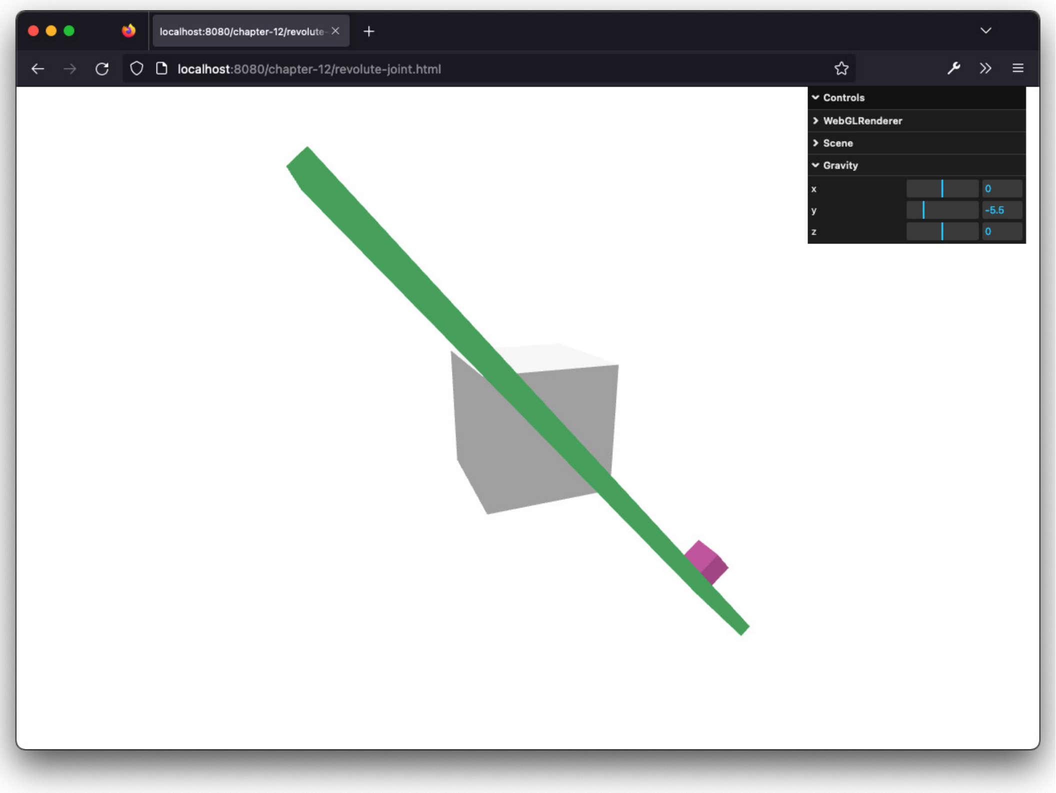

localhost:8080/chapter-12/revolute
localhost:8080/chapter-12/revolute-joint.html
Controls
WebGLRenderer
Scene
Gravity
x
0
y
-5.5
z
0

Figure 12.14 – The bar responding to the weight on one end

For a revolute joint to work, we once again need two rigid bodies. The Rapier part of the gray cube is defined like this:

```
const bodyDesc = new RAPIER.RigidBodyDesc(RigidBodyType.Fixed)
const body = world.createRigidBody(bodyDesc)
const colliderDesc = RAPIER.ColliderDesc.cuboid(0.5, 0.5, 0.5)
const collider = world.createCollider(colliderDesc, body)
```

This means, that this `RigidBody` will always be in the same location regardless of any forces exerted on it. The green bar is defined like this:

```
Const bodyDesc = new RAPIER.RigidBodyDesc
  (RigidBodyType.Dynamic)
  .setCanSleep(false)
  .setTranslation(-1, 0, 0)
  .setAngularDamping(0.1)
const body = world.createRigidBody(bodyDesc)
const colliderDesc = RAPIER.ColliderDesc.cuboid(0.25, 0.05,
  2)
const collider = world.createCollider(colliderDesc, body)
```

Nothing special here, but we have introduced a new property could `angularDamping`. With angular damping, Rapier will slowly decrease the rotation speed of a `RigidBody`. In our example, this means that the bar will slowly stop rotating after a while.

And the box that we're dropping looks like this:

```
Const bodyDesc = new RAPIER.RigidBodyDesc
  (RigidBodyType.Dynamic)
  .setCanSleep(false)
  .setTranslation(-1, 1, 1)
const body = world.createRigidBody(bodyDesc)
const colliderDesc = RAPIER.ColliderDesc.cuboid
  (0.1, 0.1, 0.1)
const collider = world.createCollider(colliderDesc, body)
```

So, at this point, we have defined `RigidBody`. Now, we can connect the fixed box with the green bar:

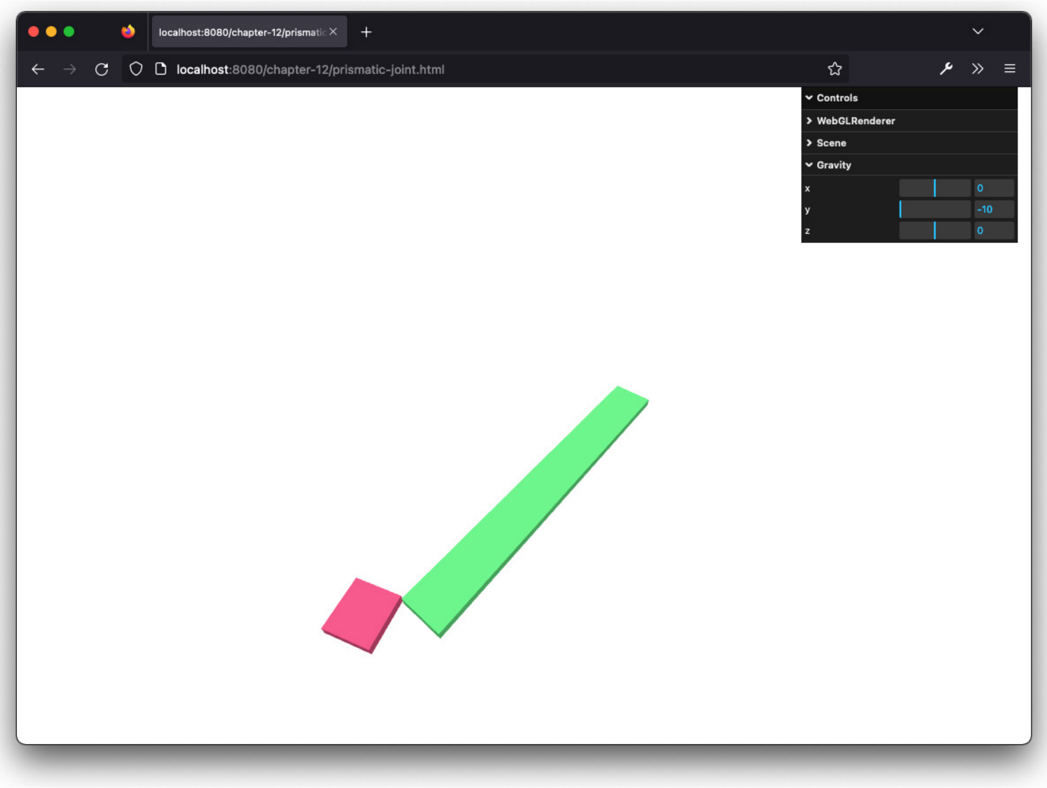
```
const params = RAPIER.JointData.revolute(
  { x: 0.0, y: 0, z: 0 },
  { x: 1.0, y: 0, z: 0 },
  { x: 1, y: 0, z: 0 }
)
let joint = world.createImpulseJoint(params, fixedCubeBody,
  greenBarBody, true)
```

The first two parameters determine the position at which the two rigid bodies are connected (following the same idea as with the fixed joint). The last parameter defines the vector at which the bodies can rotate in relation to one another. Since our first `RigidBody` is fixed, only the green bar can rotate.

The last joint type supported by Rapier is the prismatic joint.

Limiting movement to a single axis with a prismatic joint

The prismatic joint limits the movement of an object to a single axis. This is demonstrated in the following example (`prismatic.html`), where the movement of the reddish cube is limited to a single axis:


localhost:8080/chapter-12/prismatic
localhost:8080/chapter-12/prismatic-joint.html
Controls
WebGLRenderer
Scene
Gravity

x 0
y -10
z 0

Figure 12.15 – Red cube is limited to one axis

In this example, we throw a cube at the green bar with the revolute joint from the previous example. This will cause the green bar to spin around its *y*-axis at the center and hit the reddish cube. This cube is limited to movement along a single axis, and you'll see it moving along that axis.

To create the joint for this example, we used the following piece of code:

```
const prismaticParams = RAPIER.JointData.prismatic(
  { x: 0.0, y: 0.0, z: 0 },
  { x: 0.0, y: 0.0, z: 3 },
  { x: 1, y: 0, z: 0 }
)
```

```
prismaticParams.limits = [-2, 2]
prismaticParams.limitsEnabled = true
world.createImpulseJoint(prismaticParams, fixedCubeBody,
  redCubeBody, true)
```

We once again define the position (`{ x: 0.0, y: 0.0, z: 0 }`) of `fixedCubeBody` first, which defines the object we're moving in relation to. Then, we define the position of our cube – `{ x: 0.0, y: 0.0, z: 3 }`. Finally, we define the axis along which our object is allowed to move. In this case, we defined `{ x: 1, y: 0, z: 0 }`, which means it is allowed to move along its *x*-axis.

Using joint motors to move objects around their allowed axis

Spherical, revolute, and prismatic joints also support something called motors. With motors, you can move a rigid body along its allowed axis. We haven't shown this in these examples but by using motors, you can add gears that move around automatically or create a car where you move the wheels using revolute joints with the help of a motor. For more information on motors, see the relevant section of the Rapier documentation here: `https://rapier.rs/docs/user_guides/javascript/joints#joint-motors`.

As we mentioned in the *Creating a basic Three.js scene with Rapier* section, we've only scratched the surface of what is possible with Rapier. Rapier is an extensive library with many features that allow fine-tuning and should provide support for most cases in which you might need a physics engine. The library is actively being developed, and the online documentation is very good.

With the examples in this chapter and the online documentation, you should be able to integrate Rapier into your own scenes even for the features not explained in this chapter.

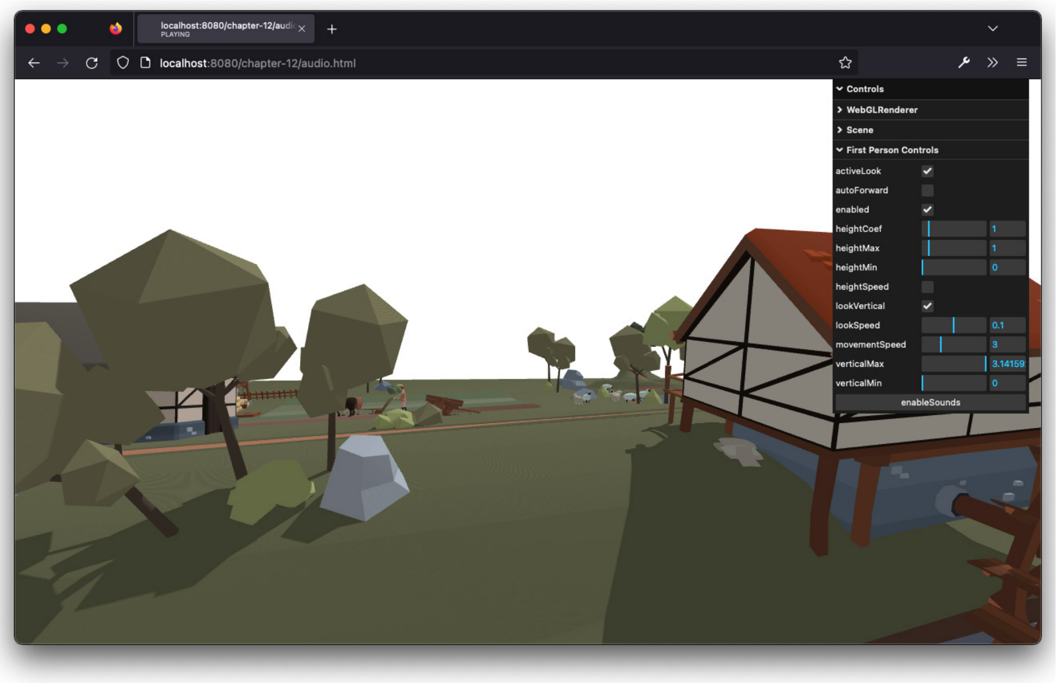
We've mainly looked at 3D models and how to render them in Three.js. However, Three.js also provides support for 3D sounds. In the next section, we'll show you an example of how you can add directional sound to a Three.js scene.

Adding sound sources to your scene

Having discussed several relevant topics by now, we have a lot of the ingredients in place to create beautiful scenes, games, and other 3D visualizations. What we haven't shown, however, is how to add sounds to your Three.js scene. In this section, we'll look at two Three.js objects that allow you to add sources of sound to your scene. This is especially interesting since these sound sources respond to the position of the camera:

- The distance between the sound source and the camera determines the volume of the sound source
- The positions of the left-hand side and the right-hand side of the camera determine the sound volume of the left-hand side speaker and the right-hand side speaker, respectively

The best way to explain this is to see this in action. Open up the `audio.html` example in your browser, and you'll see a scene from *Chapter 9*, *Animations and Moving the Camera*:


localhost:8080/chapter-12/audi
PLAYING
localhost:8080/chapter-12/audio.html
Controls
WebGLRenderer
Scene
First Person Controls
enableSounds

activeLook ☑
autoForward ☐
enabled ☑
heightCoef 1
heightMax 1
heightMin 0
heightSpeed ☐
lookVertical ☑
lookSpeed 0.1
movementSpeed 3
verticalMax 3.14159
verticalMin 0

Figure 12.16 – A scene with audio elements

This example uses the first-person controls we saw in *Chapter 9*, so you can use the arrow keys in combination with the mouse to move around the scene. Since browsers don't support starting audio automatically anymore, first, hit the `enableSounds` button in the menu on the right to turn on the sounds. When you do this, you'll hear water coming from somewhere nearby – and you'll be able to hear some cows and some sheep in the distance.

The water sounds come from the water wheel behind your starting position, the sounds from the sheep come from the flock of sheep to the right, and the cow sounds are centered on the two oxen pulling the plow. If you use the controls to move around the scene, you'll notice that the sounds change based on where you are – the nearer you get to the sheep, the better you'll hear them, and as you move to the left, the sounds of the oxen will be louder. This is something called positional audio, where the volume and direction are used to determine how to play the sounds.

Accomplishing this only takes a small amount of code. The first thing we need to do is define a `THREE.AudioListener` object and add it to `THREE.PerspectiveCamera`:

```
const listener = new THREE.AudioListener(); camera.
add(listener1);
```

Next, we need to create a `THREE.Mesh` (or a `THREE.Object3D`) instance and add a `THREE.PositionalAudio` object to that mesh. This will determine the source location of this specific sound:

```
const mesh1 = new THREE.Mesh(new THREE.BoxGeometry(1, 1,
  1), new THREE.MeshNormalMaterial({ visible: false }))
mesh1.position.set(-4, -2, 10)
scene.add(mesh1)
const posSound1 = new THREE.PositionalAudio(listener)
const audioLoader = new THREE.AudioLoader()
audioLoader.load('/assets/sounds/water.mp3', function
  (buffer) {
posSound1.setBuffer(buffer)
posSound1.setRefDistance(1)
posSound1.setRolloffFactor(3)
posSound1.setLoop(true)
mesh1.add(posSound3)
```

As you can see from this code snippet, we first create a standard `THREE.Mesh` instance. Next, we create a `THREE.PositionalAudio` object, which we connect to the `THREE.AudioListener` object that we created earlier. Finally, we add the audio and configure some properties, which define how the sound is played and how it behaves:

- `setRefDistance`: This determines the distance from the object at which the sound will be reduced in volume.
- `setLoop`: By default, a sound is played once. By setting this property to `true`, the sound loops.
- `setRolloffFactor`: This determines how quickly the volume decreases as you move away from the sound source.

Internally, Three.js uses the Web Audio API (`http://webaudio.github.io/web-audio-api/`) to play the sound and determine the correct volume. Not all browsers support this specification. The best support currently is from Chrome and Firefox.

Summary

In this chapter, we explored how you can extend the basic 3D functionality of Three.js by adding physics. For this, we used the Rapier library, which allows you to add gravity to your scene and objects, have objects interact with each other and bounce when they collide, and use joints to limit the movement of objects relative to each other.

Besides that, we also showed you how Three.js supports 3D sounds. We created a scene where you added positional sound using the `THREE.PositionalAudio` and `THREE.AudioListener` objects.

Even though we've now covered all of the core functionalities provided by Three.js, there are two more chapters dedicated to exploring some external tools and libraries that you can use together with Three.js. In the next chapter, we'll dive into Blender and see how we can use Blender's functionality, such as baking shadows, editing UV maps, and exchanging models between Blender and Three.js.

13

Working with Blender and Three.js

In this chapter, we'll dive a bit deeper into how you can use Blender and Three.js together. We'll explain the following concepts in this chapter:

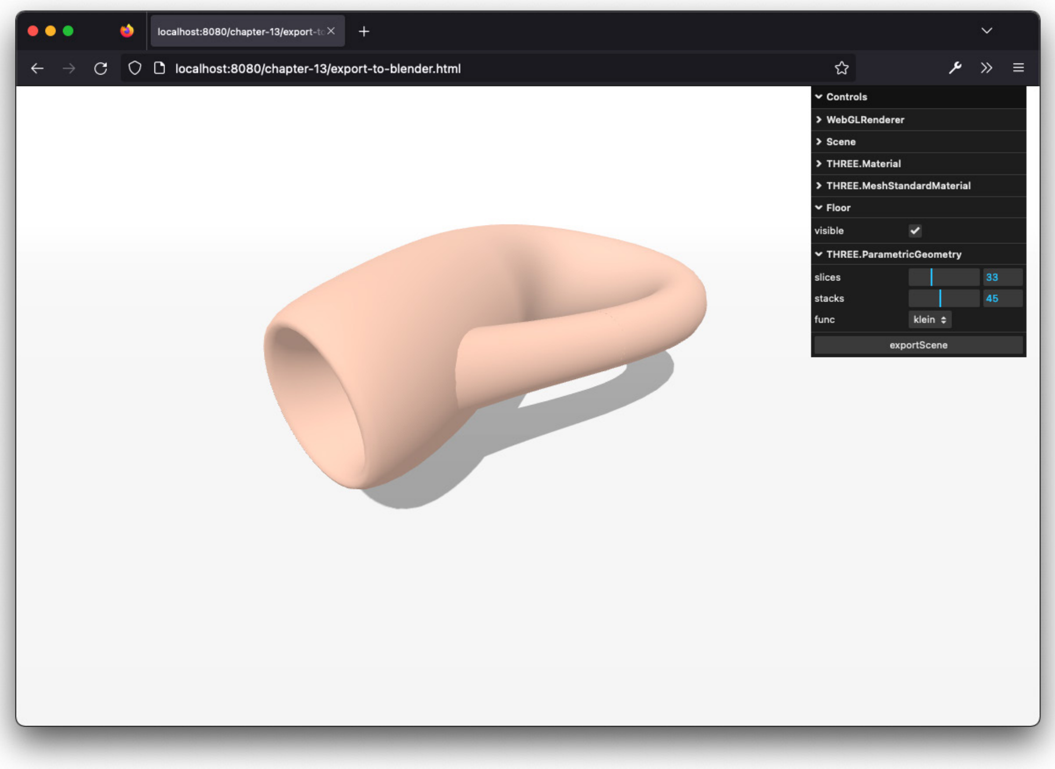
- *Exporting from Three.js and importing into Blender*: We'll create a simple scene, export it from Three.js, and load and render it in Blender.
- *Exporting a static scene from Blender and importing it into Three.js*: Here, we will create a scene in Blender, export it into Three.js, and render it in Three.js.
- *Exporting an animation from Blender and importing it into Three.js*: Blender allows us to create animations, we'll create a simple animation, and load and show it in Three.js.
- *Baking lightmaps and ambient occlusion maps in Blender*: Blender allows us to bake different types of maps that we can use in Three.js.
- *Custom UV modeling in Blender*: With UV modeling, we determine how a texture is applied to a geometry. Blender provides a lot of tools to make that easy. We'll explore how you can use the UV modeling capabilities of Blender and use the results in Three.js.

Before we get started with this chapter, make sure to install Blender so that you can follow along. You can install Blender by downloading the installer for your OS from here: `https://www.blender.org/download/`. The screenshots shown of Blender in this chapter were taken using the macOS version of Blender, but the versions for Windows and Linux look the same.

Let's get started with our first topic, where we create a scene in Three.js, export it to an intermediate format, and finally import it into Blender.

Exporting from Three.js and importing into Blender

For this example, we'll just take a simple sample reusing the parametric geometry we saw in *Chapter 6*, *Exploring Advanced Geometries*. If you open `export-to-blender.html` in the browser, you can create some parametric geometries. At the bottom of the menu on the right, we've added an **exportScene** button:


localhost:8080/chapter-13/export-t
localhost:8080/chapter-13/export-to-blender.html
Controls
WebGLRenderer
Scene
THREE.Material
THREE.MeshStandardMaterial
Floor
THREE.ParametricGeometry
exportScene

visible ✓

slices 33

stacks 45

func klein

Figure 13.1 – A simple scene that we'll export

When you click on that button, the model will be saved in the GLTF format and downloaded onto your computer. To export a model with Three.js, we can use `GLTFexporter` like so:

```
const exporter = new GLTFExporter()
const options = {
  trs: false,
  onlyVisible: true,
  binary: false
}
```

```
exporter.parse(
  scene,
  (result) => {
    const output = JSON.stringify(result, null, 2)
    save(new Blob([output], { type: 'text/plain' }),
      'out.gltf')
  },
  (error) => {
    console.log('An error happened during parsing of the
      scene', error)
  },
  options
)
```

Here, you can see that we have created a `GLTFExporter` that we can use to export a `THREE.Scene`. We can export a scene in the glTF binary format or the JSON format. For this example, we export in JSON. The glTF format is a complex format, and while `GLTFExporter` supports many of the objects that make up a Three.js scene, you can still run into issues where the export fails. Updating to the latest version of Three.js is often the best solution since work is being constantly done on this component.

Once we've got our `output`, we can trigger the browser's `download` functionality, which will save it to your local machine:

```
const save = (blob, filename) => {
  const link = document.createElement('a')
  link.style.display = 'none'
  document.body.appendChild(link)
  link.href = URL.createObjectURL(blob)
  link.download = filename
  link.click()
}
```

The result is a glTF file, and its first couple of lines look like this:

```
{
  "asset": {
    "version": "2.0",
     "generator": "THREE.GLTFExporter"
   },
   "scenes": [
     {
       "nodes": [
         0,
         1,
         2,
         3
       ]
     }
   ],
   "scene": 0,
   "nodes": [
     {},
...
```

Now that we've got a glTF file containing our scene, we can import this into Blender. So, open up Blender and you'll be presented with the default scene with a single cube. Remove the cube by selecting it and pressing **x**. Once removed, we have an empty scene in which we'll load our exported scene.

From the **File** menu at the top, select **Import | glTF 2.0**, and you'll be presented with a file browser. Navigate to where you've downloaded the model, select the file, and click on **Import glTF 2.0**. This will open the file, and show you something like this:

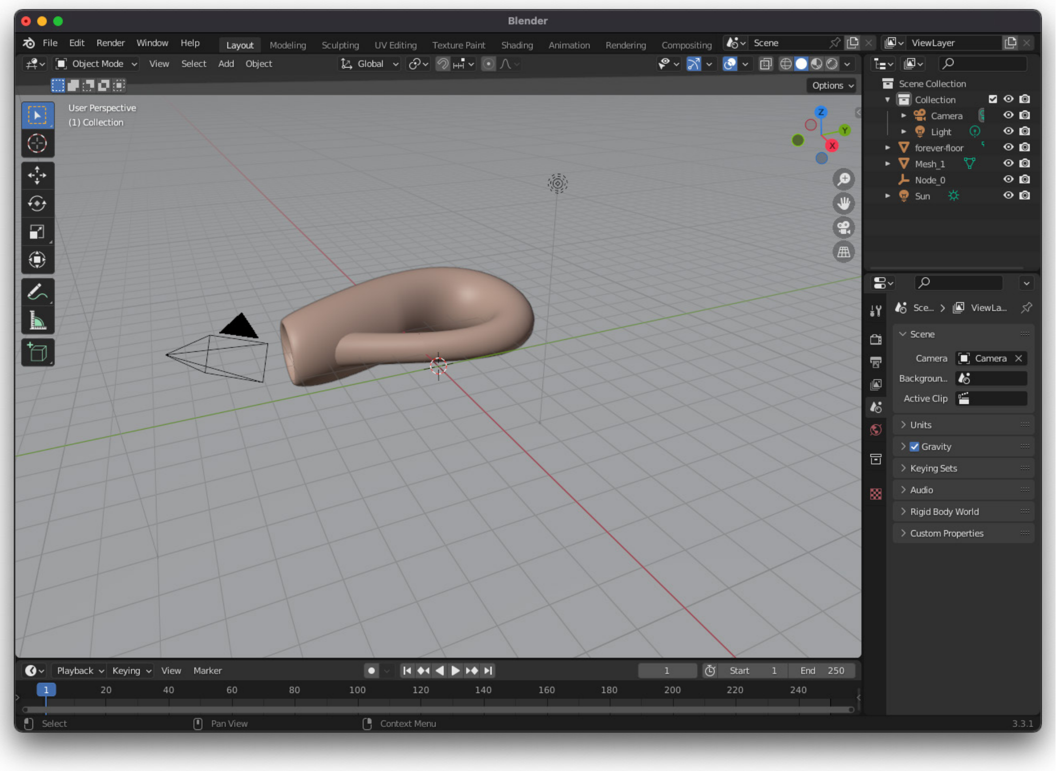
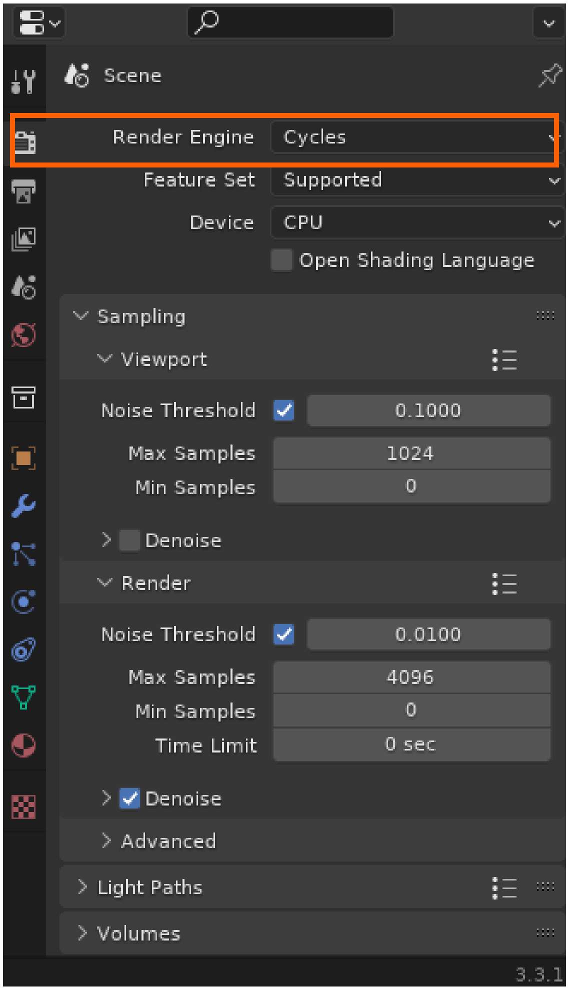

Blender
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(1) Collection
Scene Collection
Collection
Camera
Light
forever-floor
Mesh_1
Node_0
Sun
Sce...
ViewLa...
Scene
Camera
Camera
Backgroun...
Active Clip
Units
Gravity
Keying Sets
Audio
Rigid Body World
Custom Properties
Playback
Keying
View
Marker
1
Start 1
End 250
1
20
40
60
80
100
120
140
160
180
200
220
240
Select
Pan View
Context Menu
3.3.1

Figure 13.2 – Three.js scene imported in Blender

As you can see, Blender has imported our complete scene, and the `THREE.Mesh` we defined in Three.js is now available in Blender. In Blender, we can now use this just like any other mesh. For this example, however, let's keep it simple and just render this scene with the **Cycles** Blender renderer. To do this, click on **Render Properties** in the menu on the right (the icon that looks like a camera) and for **Render Engine**, select **Cycles**:


Scene
Sampling
Viewport
Render
Advanced
Light Paths
Volumes
3.3.1

Render Engine: Cycles
Feature Set: Supported
Device: CPU
☐ Open Shading Language

Viewport
Noise Threshold ☑ 0.1000
Max Samples 1024
Min Samples 0
☐ Denoise

Render
Noise Threshold ☑ 0.0100
Max Samples 4096
Min Samples 0
Time Limit 0 sec
☑ Denoise

Figure 13.3 – Rendering with the Cycles render engine in Blender

Next, we need to position the camera correctly, so use the mouse to move around the scene until you've got a view you're happy with, and then press *Ctrl* + *Alt* + *numpad 0* to align the camera. At this point, you'll have something like this:

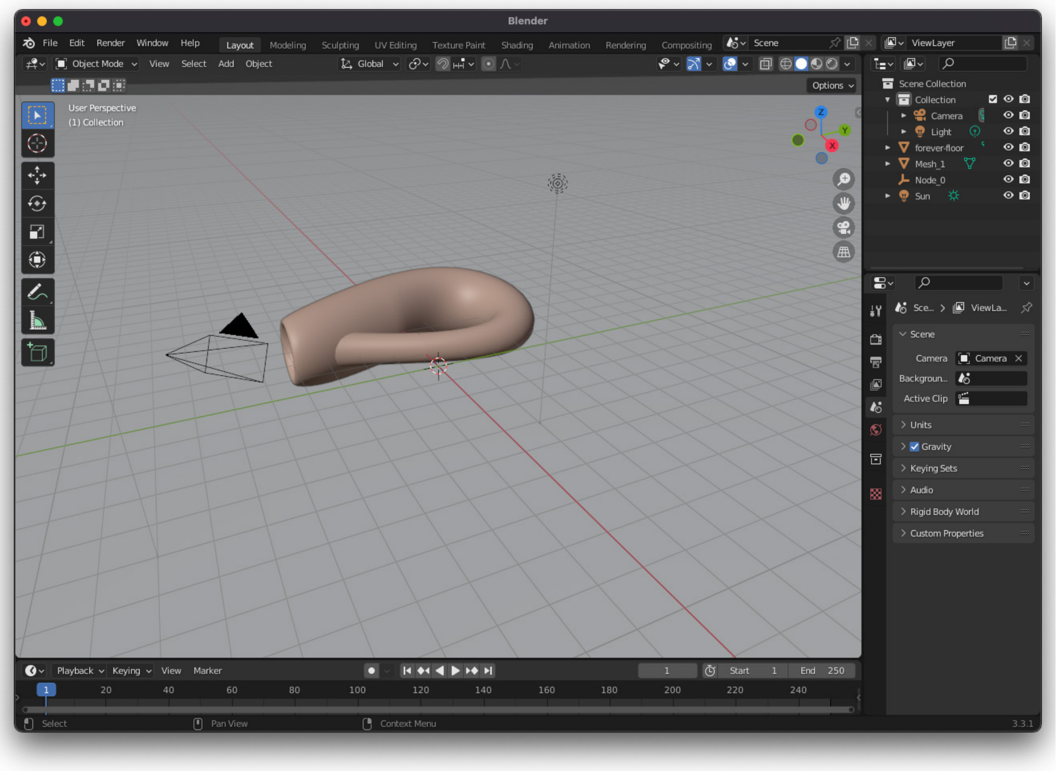

Blender
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(1) Collection
Scene Collection
Collection
Camera
Light
forever-floor
Mesh_1
Node_0
Sun
Sce...
ViewLa...
Scene
Camera
Camera
Backgroun...
Active Clip
Units
Gravity
Keying Sets
Audio
Rigid Body World
Custom Properties
Playback
Keying
View
Marker
Start 1
End 250
Select
Pan View
Context Menu
3.3.1

Figure 13.4 – Showing the area the camera sees and what will be rendered

Now, we can render the scene by hitting *F12*. This will start the **Cycles** render engine, and you'll see the model being rendered in Blender:

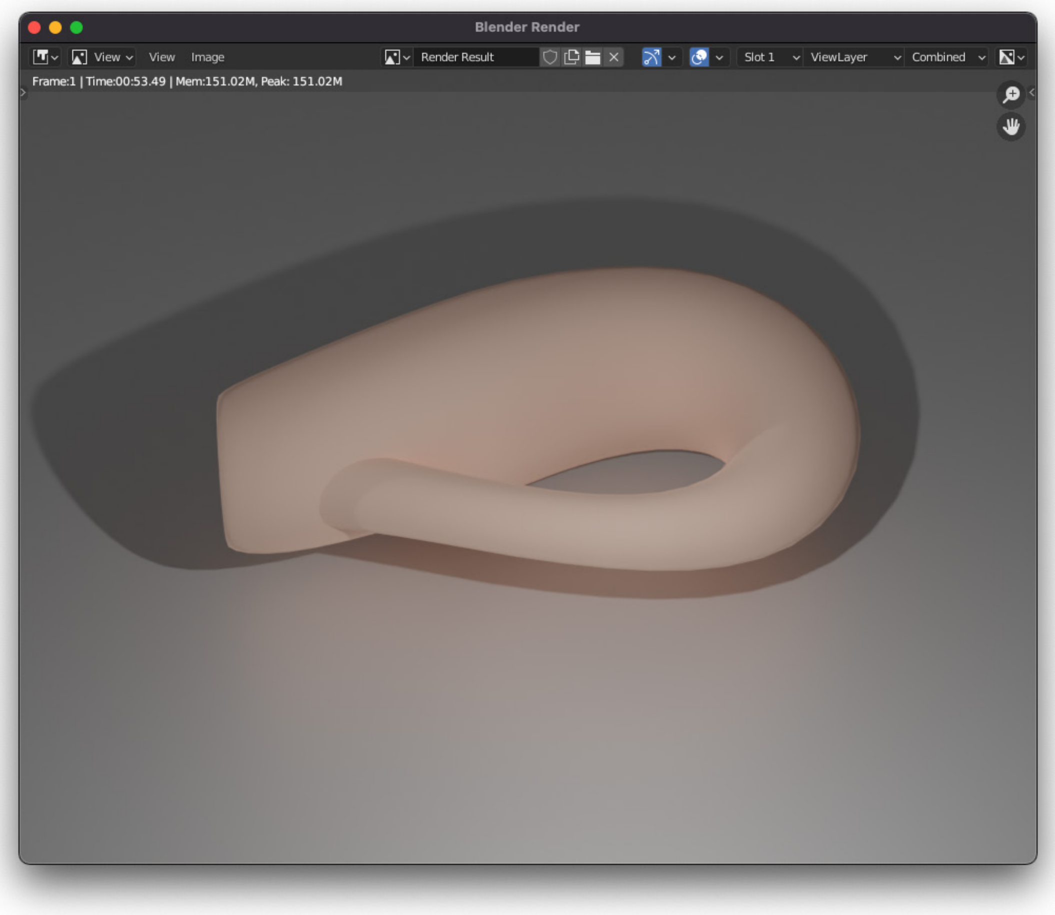

Blender Render
View
View
Image
Render Result
Slot 1
ViewLayer
Combined
Frame:1 | Time:00:53.49 | Mem:151.02M, Peak: 151.02M

Figure 13.5 – Final image being rendered in Blender from our exported Three.js model

As you've seen, using the glTF as a format for exchanging models and scenes between Three.js and Blender is very straightforward. Just use `GLTFExporter`, import the model in Blender, and you can use everything Blender has to offer on your model.

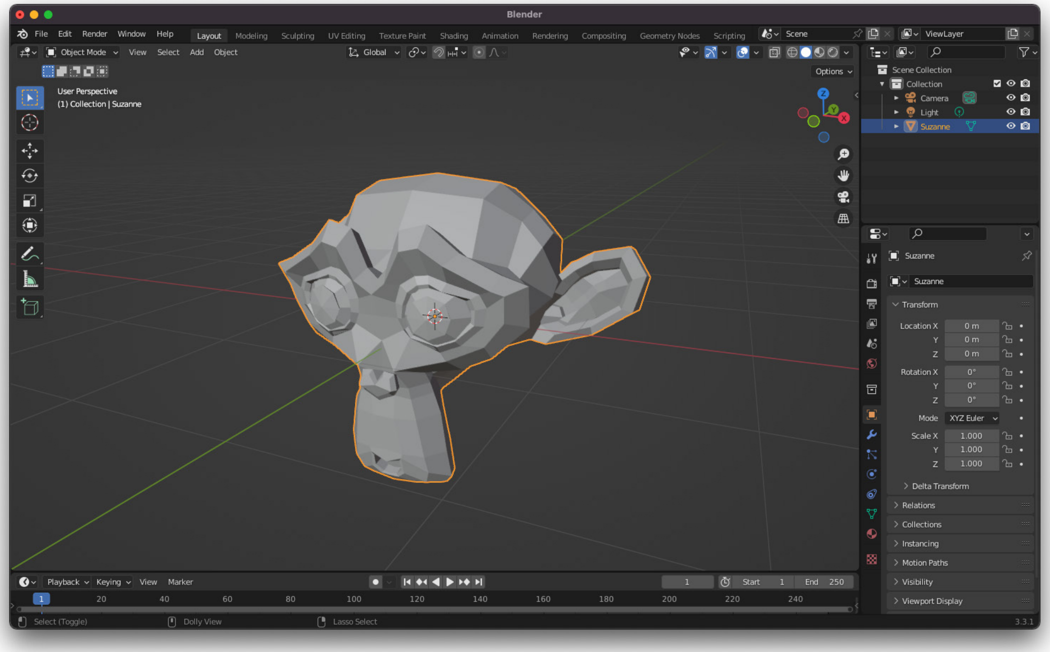
Of course, the other way around works just as easily, as we'll show you in the next section.

Exporting a static scene from Blender and importing it into Three.js

Exporting models from Blender is just as easy as importing them. In the older version of Three.js, there was a specific Blender plugin you could use to export in a Three.js-specific JSON format. In later versions though, glTF in Three.js has become the standard for exchanging models with other tools. So, to get this working with Blender, all we have to do is this:

1. Create a model in Blender.
2. Export the model to a glTF file.
3. Import the glTF file in Blender and add it to the scene.

Let's create a simple model in Blender first. We'll use the default model Blender uses, which can be added in **Object Mode** by selecting **Add | Mesh | Monkey** from the menu. Click on **monkey** to select it:


Blender
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes Scripting
Scene
ViewLayer
Object Mode View Select Add Object
Global
Options
User Perspective
(1) Collection | Suzanne
Scene Collection
Collection
Camera
Light
Suzanne
Playback Keying View Marker
Start 1 End 250
Select (Toggle) Dolly View Lasso Select
3.3.1

Suzanne

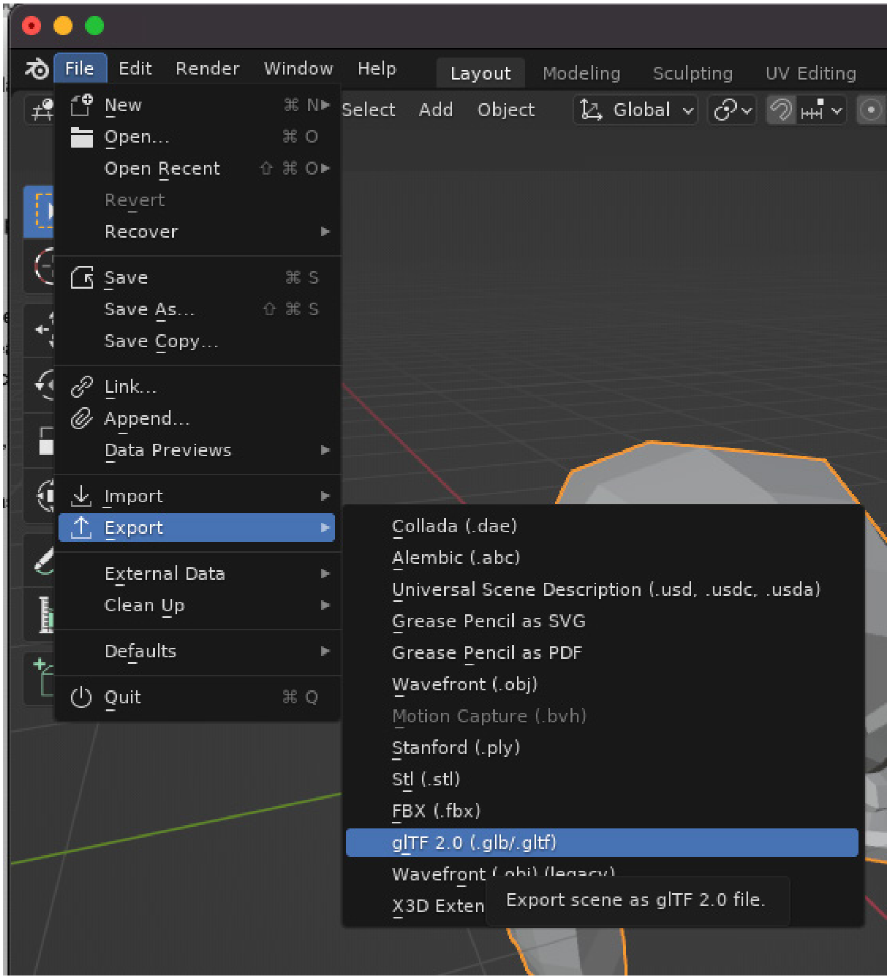
Transform

Location X 0 m
Y 0 m
Z 0 m
Rotation X 0°
Y 0°
Z 0°
Mode XYZ Euler
Scale X 1.000
Y 1.000
Z 1.000

Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display

Figure 13.6 – Creating the model in Blender that you want to export

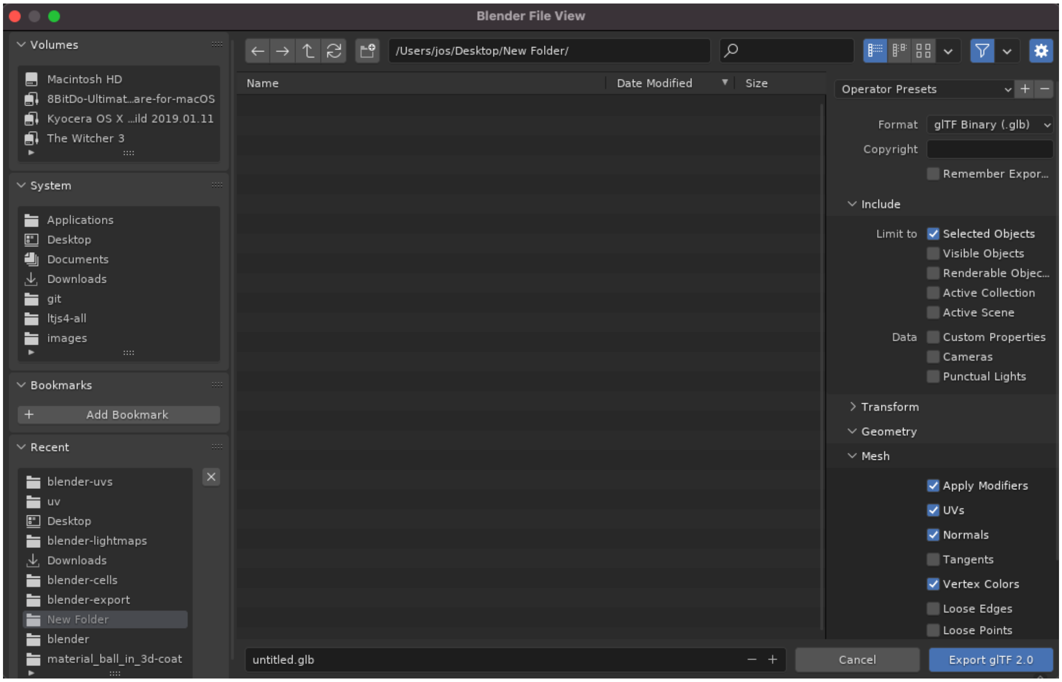
Once the model is selected, in the top menu, select `File->Export->glTF 2.0`:


File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Select
Add
Object
Global
New ⌘ N
Open... ⌘ O
Open Recent ⇧ ⌘ O
Revert
Recover
Save ⌘ S
Save As... ⇧ ⌘ S
Save Copy...
Link...
Append...
Data Previews
Import
Export
External Data
Clean Up
Defaults
Quit ⌘ Q
Collada (.dae)
Alembic (.abc)
Universal Scene Description (.usd, .usdc, .usda)
Grease Pencil as SVG
Grease Pencil as PDF
Wavefront (.obj)
Motion Capture (.bvh)
Stanford (.ply)
Stl (.stl)
FBX (.fbx)
glTF 2.0 (.glb/.gltf)
Wavefront (.obj) (legacy)
X3D Exten

Export scene as glTF 2.0 file.

Figure 13.7 – Selecting the glTF export

For this example, we only export the mesh. Note that when you're exporting from Blender, always check the **Apply Modifiers** checkbox. This will make sure any advanced generators or modifiers used in Blender are applied before exporting the mesh.


Blender File View
Volumes
Macintosh HD
8BitDo-Ultimat...are-for-macOS
Kyocera OS X ...ild 2019.01.11
The Witcher 3
System
Applications
Desktop
Documents
Downloads
git
ltjs4-all
images
Bookmarks
Add Bookmark
Recent
blender-uvs
uv
Desktop
blender-lightmaps
Downloads
blender-cells
blender-export
New Folder
blender
material_ball_in_3d-coat
Name
Date Modified
Size
Operator Presets
Include
Transform
Geometry
Mesh
Cancel
Export glTF 2.0

/Users/jos/Desktop/New Folder/

Format: glTF Binary (.glb)

Copyright:

☐ Remember Expor...

Limit to: ☑ Selected Objects, ☐ Visible Objects, ☐ Renderable Objec..., ☐ Active Collection, ☐ Active Scene

Data: ☐ Custom Properties, ☐ Cameras, ☐ Punctual Lights

Mesh: ☑ Apply Modifiers, ☑ UVs, ☑ Normals, ☐ Tangents, ☑ Vertex Colors, ☐ Loose Edges, ☐ Loose Points

untitled.glb

Figure 13.8 – Exporting the model as a glTF file

Once the file is exported, we can load it in Three.js using `GLTFImporter`:

```
const loader = new GLTFLoader()
return loader.loadAsync('/assets/gltf/
  blender-export/monkey.glb').then((structure) => {
  return structure.scene
})
```

The final result is the exact model from Blender, but visualized in Three.js (see the `import-from-blender.html` example):

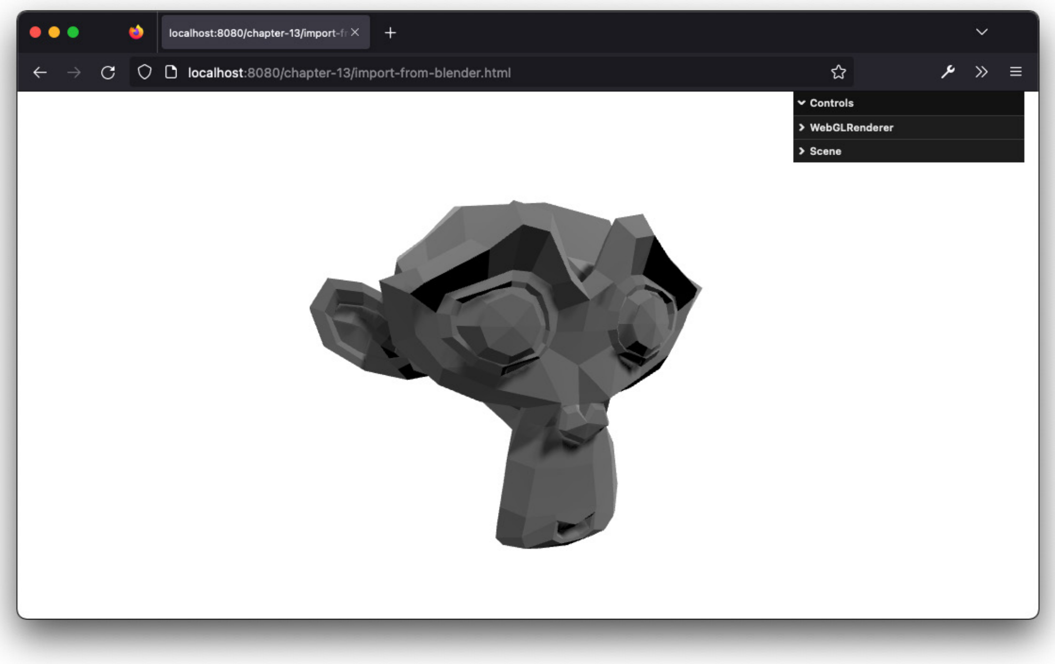
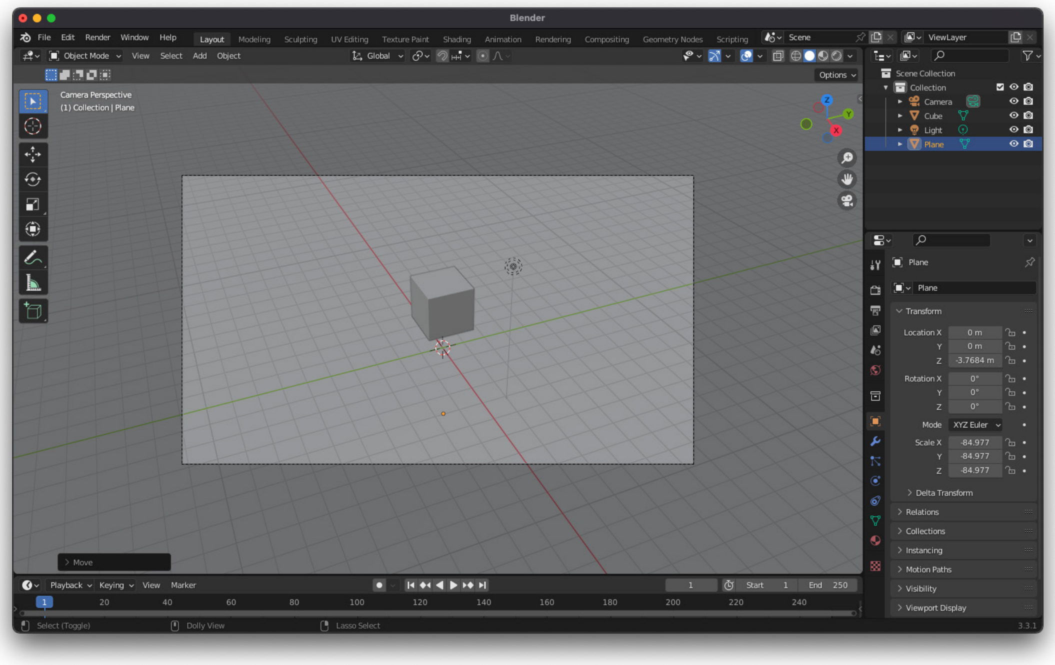

localhost:8080/chapter-13/import-fr
localhost:8080/chapter-13/import-from-blender.html
Controls
WebGLRenderer
Scene

Figure 13.9 – The Blender model visualized in Three.js

Note that this isn't just limited to the meshes – with glTF, we can also export lights, cameras, and textures in the same manner.

Exporting an animation from Blender and importing it into Three.js

Exporting an animation from Blender works in pretty much the same way as exporting a static scene. Therefore, for this example, we'll create a simple animation, export it in the glTF format again, and load it into a Three.js scene. For this, we're going to create a simple scene where we render a cube falling and breaking into parts. The first thing we need for this is a floor and a cube. Therefore, create a plane and a cube that hangs a little bit above this plane:


Blender
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
Camera Perspective
(1) Collection | Plane
Scene Collection
Collection
Camera
Cube
Light
Plane
Transform
Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display
Move
Playback
Keying
View
Marker
Start 1
End 250
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Plane

Location X 0 m
Y 0 m
Z -3.7684 m

Rotation X 0°
Y 0°
Z 0°

Mode XYZ Euler

Scale X -84.977
Y -84.977
Z -84.977

Figure 13.10 – An empty Blender project

Here, we just moved the cube up a little bit (press *G* to grab the cube) and added a plane (**Add | Mesh | Plane**), and then we scaled this plane to make it bigger. Now, we can add physics to the scene. In *Chapter 12*, *Adding Physics and Sounds to Your Scene*, we introduced the concept of rigid bodies. Blender uses this same approach. Select the cube and use **Object | Rigid Body | Add Active**, and select the plane and add its rigid body like this: **Object | Rigid Body | Add Passive**. At this point, when we play (by using the *spacebar*) the animation in Blender, you'll see that the cube falls:

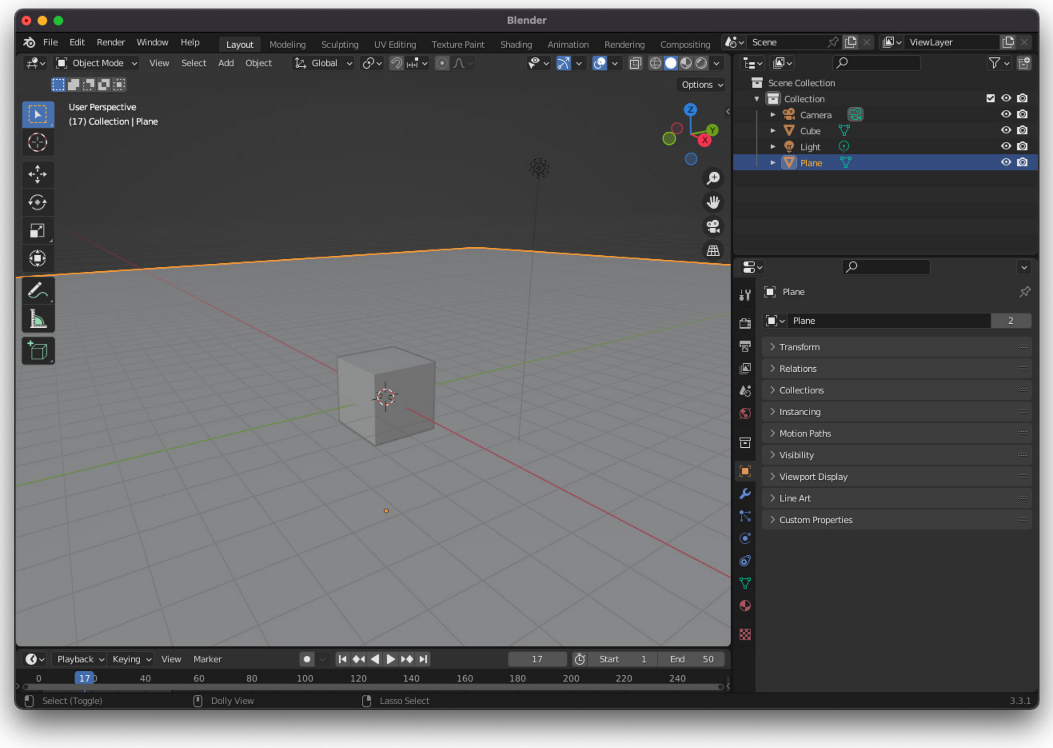
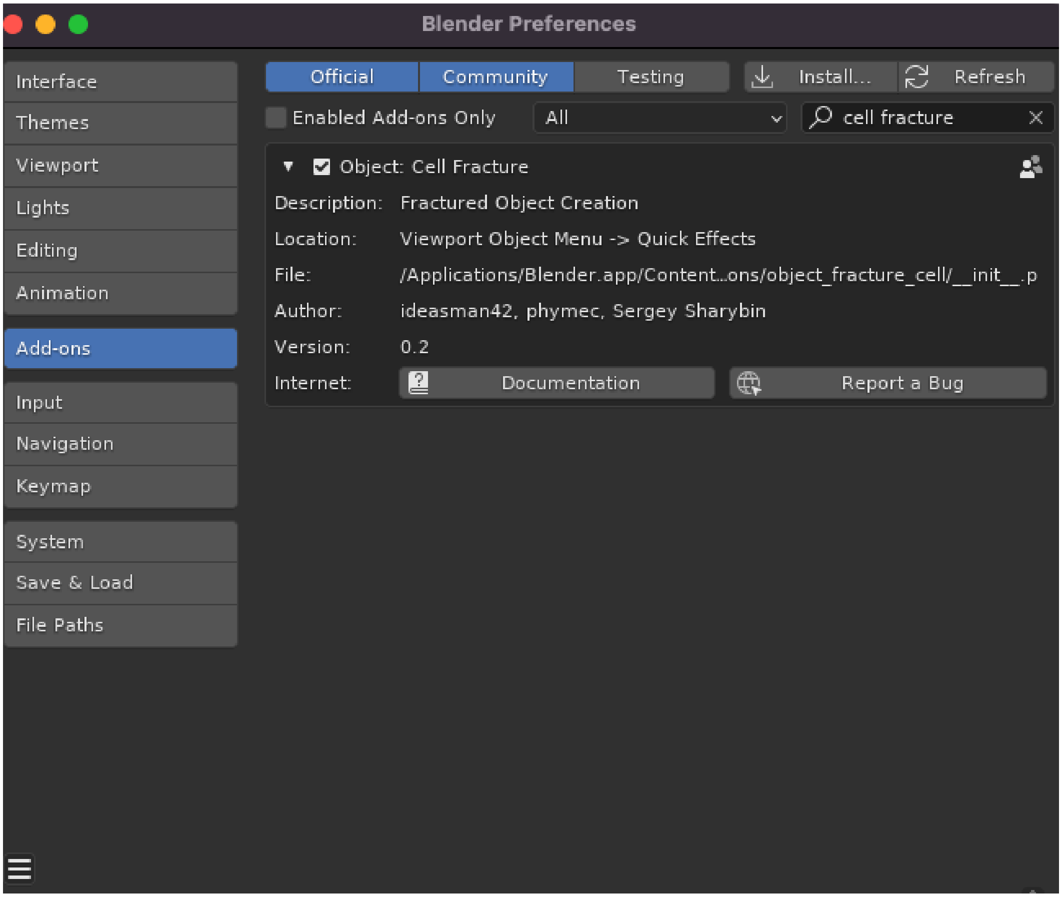

Blender
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(17) Collection | Plane
Scene Collection
Collection
Camera
Cube
Light
Plane
Plane
Plane
2
Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display
Line Art
Custom Properties
Playback
Keying
View
Marker
17
Start 1
End 50
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Figure 13.11 – A halfway animation of a cube falling

To create the breaking block effect, we need to enable the **Cell Fracture** plugin. For this, go to the **Edit | Preferences** screen, select **Add-ons**, use the search option to search for the `Cell Fracture` plugin, and check the checkbox to enable the plugin:

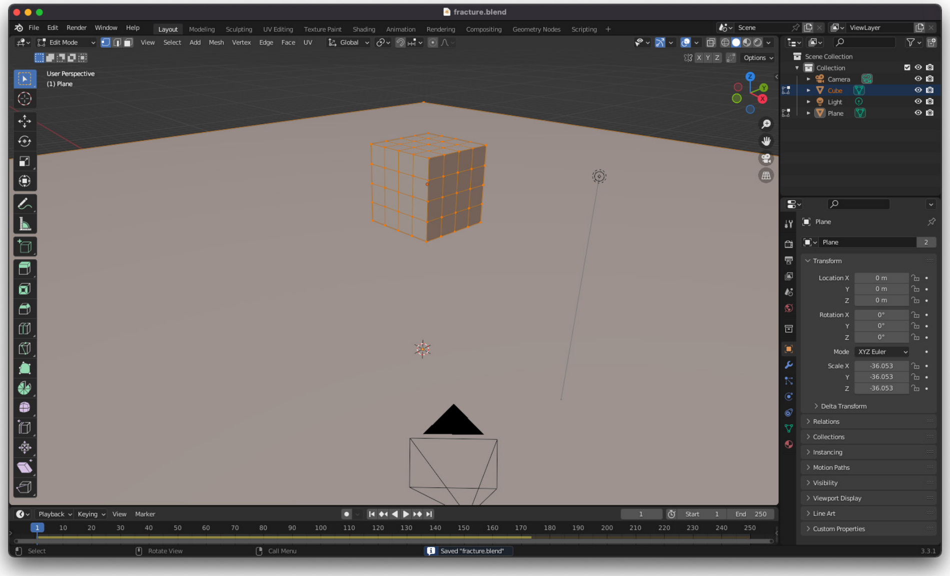

Blender Preferences
Interface
Themes
Viewport
Lights
Editing
Animation
Add-ons
Input
Navigation
Keymap
System
Save & Load
File Paths
Official
Community
Testing
Install...
Refresh
Enabled Add-ons Only
All
cell fracture
Documentation
Report a Bug

▼ ☑ Object: Cell Fracture

Description: Fractured Object Creation
Location: Viewport Object Menu -> Quick Effects
File: /Applications/Blender.app/Content...ons/object_fracture_cell/__init__.p
Author: ideasman42, phymec, Sergey Sharybin
Version: 0.2
Internet:

Figure 13.12 – Enabling the Cell Fracture plugin

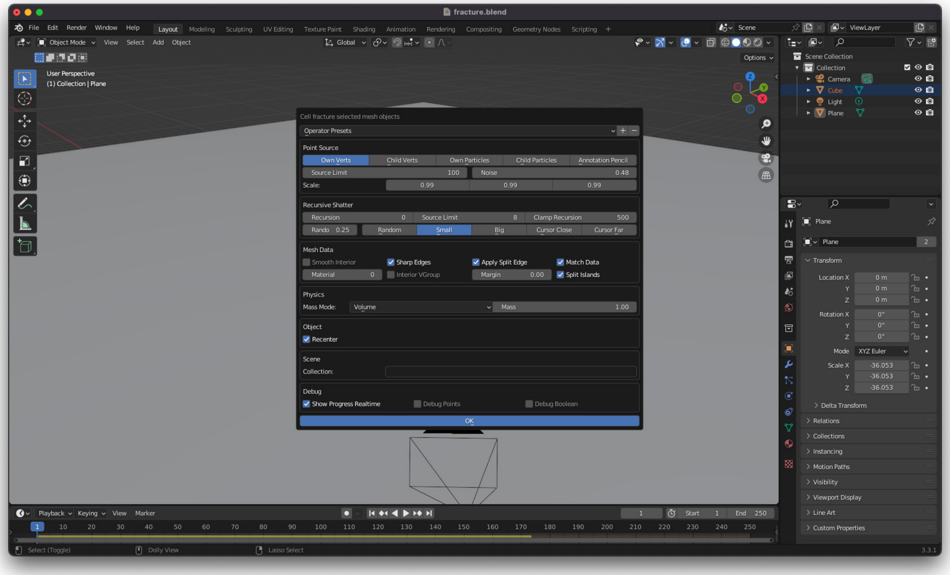
Before we split up the cube into smaller parts, let's add some vertices to the model so that Blender has a good number of vertices, which it can use to split up the model. For this, select the cube in **Edit Mode** (by using the *Tab* key) and from the menu at the top, select **Edge | Subdivide**. Do this twice, and you'll have something looking like this:


fracture.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Edit Mode
View
Select
Add
Mesh
Vertex
Edge
Face
UV
Global
Options
User Perspective
(1) Plane
Scene
ViewLayer
Scene Collection
Collection
Camera
Cube
Light
Plane
Transform
Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display
Line Art
Custom Properties
Playback
Keying
Marker
Start
End
Select
Rotate View
Call Menu
Saved "fracture.blend"

Location X 0 m
Y 0 m
Z 0 m
Rotation X 0°
Y 0°
Z 0°
Mode XYZ Euler
Scale X -36.053
Y -36.053
Z -36.053

Figure 13.13 – Showing the cube with a number of sub-divisions

Hit *Tab* to go back to **Object Mode** and with the cube selected, open the **Cell Fracture** window, and go to **Object | Quick Effects | Cell Fracture**:


fracture.blend
Object Mode
View
Select
Add
Object
User Perspective
(1) Collection | Plane
Operator Presets
OK
Select (Toggle)
Dolly View
Lasso Select

Cell fracture selected mesh objects

Point Source
Own Verts (selected) | Child Verts | Own Particles | Child Particles | Annotation Pencil
Source Limit 100
Noise 0.48
Scale: 0.99 0.99 0.99

Recursive Shatter
Recursion 0
Source Limit 8
Clamp Recursion 500
Rando 0.25
Random | Small (selected) | Big | Cursor Close | Cursor Far

Mesh Data
☐ Smooth Interior
☑ Sharp Edges
☑ Apply Split Edge
☑ Match Data
Material 0
☐ Interior VGroup
Margin 0.00
☑ Split Islands

Physics
Mass Mode: Volume
Mass 1.00

Object
☑ Recenter

Scene
Collection:

Debug
☑ Show Progress Realtime
☐ Debug Points
☐ Debug Boolean

Figure 13.14 – Configuring fractures

You can play around with these settings to get different kinds of fractures. With the settings configured in *Figure 13.3*, you'll get something like this:

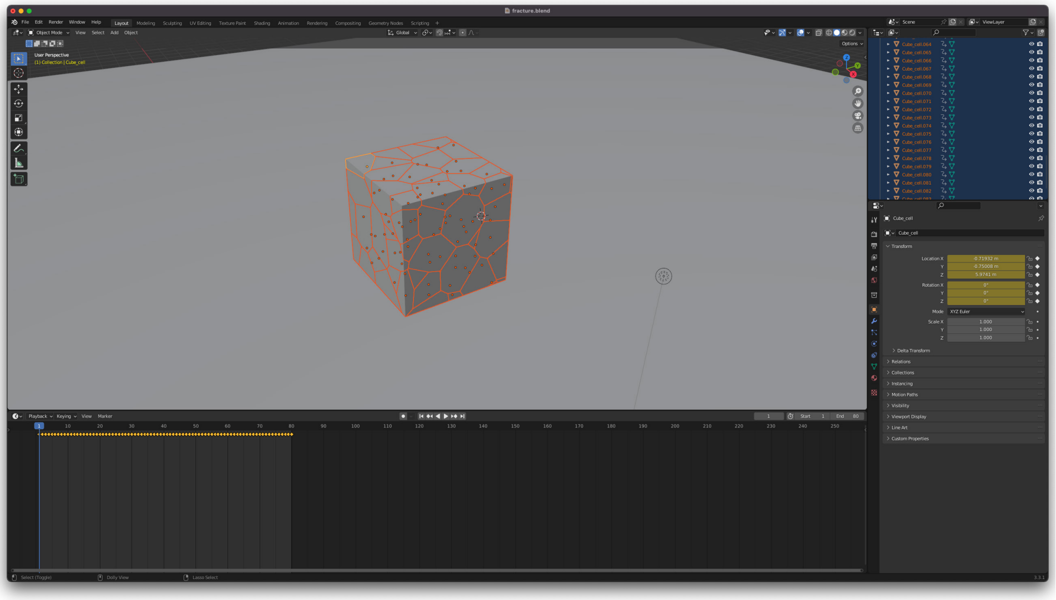
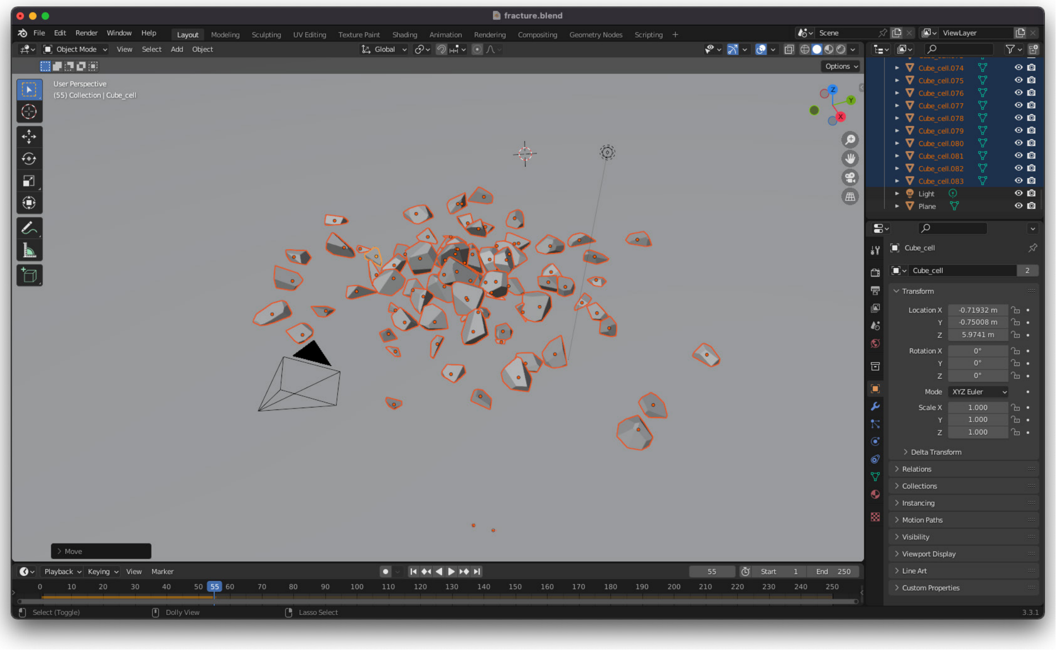


Figure 13.15 – Cube showing fractures

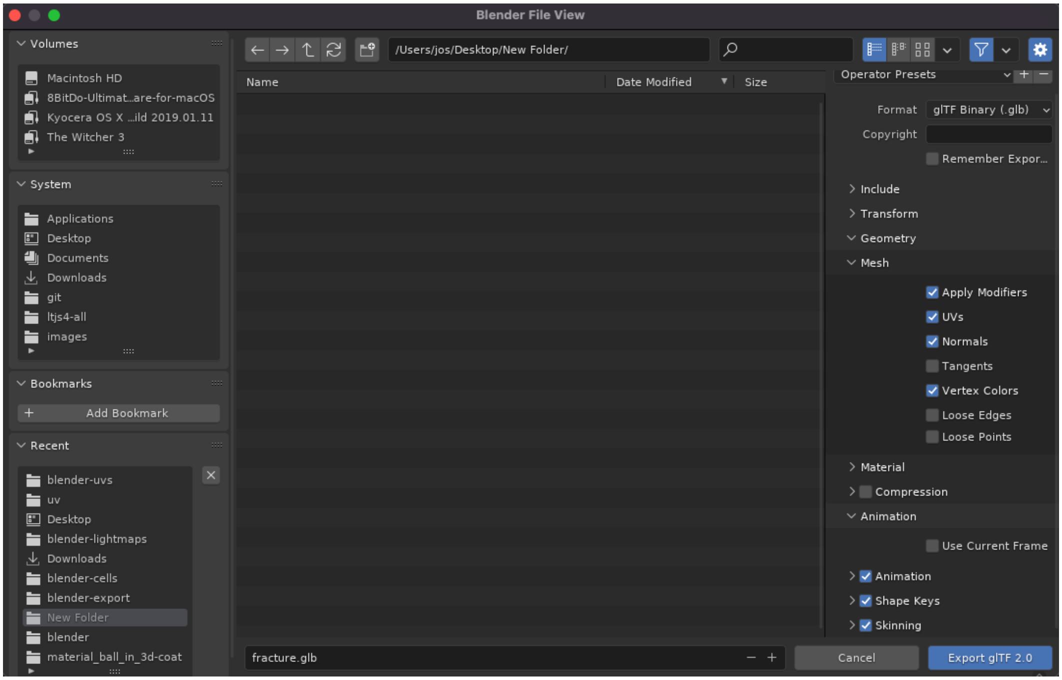
Next, select the original cube and hit **x** to delete it. This will only leave the fractured parts, which we'll animate. To do this, select all the cells from the cube and use **Object | Rigid Body | Add Active** again. Once done, hit the *spacebar* and you'll see the cube falling and breaking down on impact.


fracture.blend
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes Scripting
Scene
ViewLayer
Object Mode View Select Add Object
Global
Options
User Perspective
(55) Collection | Cube_cell
Cube_cell.074
Cube_cell.075
Cube_cell.076
Cube_cell.077
Cube_cell.078
Cube_cell.079
Cube_cell.080
Cube_cell.081
Cube_cell.082
Cube_cell.083
Light
Plane
Cube_cell
Transform
Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display
Line Art
Custom Properties
Move
Playback Keying View Marker
0 10 20 30 40 50 55 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250
Start 1 End 250
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Location X -0.71932 m
Y -0.75008 m
Z 5.9741 m
Rotation X 0°
Y 0°
Z 0°
Mode XYZ Euler
Scale X 1.000
Y 1.000
Z 1.000

Figure 13.16 – The cube after being dropped

At this point, we've pretty much got our animation ready. Now, we need to export this animation so that we can load it into Three.js and replay it from there. Before we do this, make sure to set the end of the animation (the lower-right corner of the screen) at frame 80, since it isn't that useful to export the full 250 frames. Besides this, we need to tell Blender to convert the information from the physics engine into a set of keyframes. This needs to be done since we can't export the physics engine itself, so we have to bake the position and rotation of all the meshes so that we can export them. To do this, select all the cells again, and use **Object | Rigid Body | Bake to Keyframes**. You can select the defaults and click on the **Export glTF2.0** button to get the following screen:


Blender File View
Volumes
Macintosh HD
8BitDo-Ultimat...are-for-macOS
Kyocera OS X ...ild 2019.01.11
The Witcher 3
System
Applications
Desktop
Documents
Downloads
git
ltjs4-all
images
Bookmarks
Add Bookmark
Recent
blender-uvs
uv
Desktop
blender-lightmaps
Downloads
blender-cells
blender-export
New Folder
blender
material_ball_in_3d-coat
/Users/jos/Desktop/New Folder/
Name
Date Modified
Size
Operator Presets
Cancel
Export glTF 2.0

Format glTF Binary (.glb)

Copyright

☐ Remember Expor...

Include

Transform

Geometry

Mesh

☑ Apply Modifiers
☑ UVs
☑ Normals
☐ Tangents
☑ Vertex Colors
☐ Loose Edges
☐ Loose Points

Material

☐ Compression

Animation

☐ Use Current Frame

☑ Animation
☑ Shape Keys
☑ Skinning

fracture.glb

Figure 13.17 – Animation export settings

At this point, we'll have an animation for each of the cells, which keeps track of the rotation and position of the individual meshes. With this information, we can load the scene in Three.js and set up the animation mixer for playback:

```
const mixers = []
const modelAsync = () => {
  const loader = new GLTFLoader()
  return loader.loadAsync('/assets/models/
     blender-cells/fracture.glb').then((structure) => {
    console.log(structure)
    // setup the ground plane
    const planeMesh = structure.scene.
      getObjectByName('Plane')
    planeMesh.material.side = THREE.DoubleSide
    planeMesh.material.color = new THREE.Color(0xff5555)
    // setup the material for the pieces
    const materialPieces = new THREE.MeshStandardMaterial({
color: 0xffcc33 })
    structure.animations.forEach((animation) => {
      const meshName = animation.name.substring
     (0, animation.name.indexOf('Action')).replace('.', '')
      const mesh = structure.scene.
        getObjectByName(meshName)
      mesh.material = materialPieces
      const mixer = new THREE.AnimationMixer(mesh)
      const action = mixer.clipAction(animation)
      action.play()
      mixers.push(mixer)
    })
    applyShadowsAndDepthWrite(structure.scene)
    return structure.scene
  })
}
```

In the render loop, we need to update the mixer for each animation:

```
const clock = new THREE.Clock()
const onRender = () => {
  const delta = clock.getDelta()
  mixers.forEach((mixer) => {
    mixer.update(delta)
  })
}
```

The result looks like this:

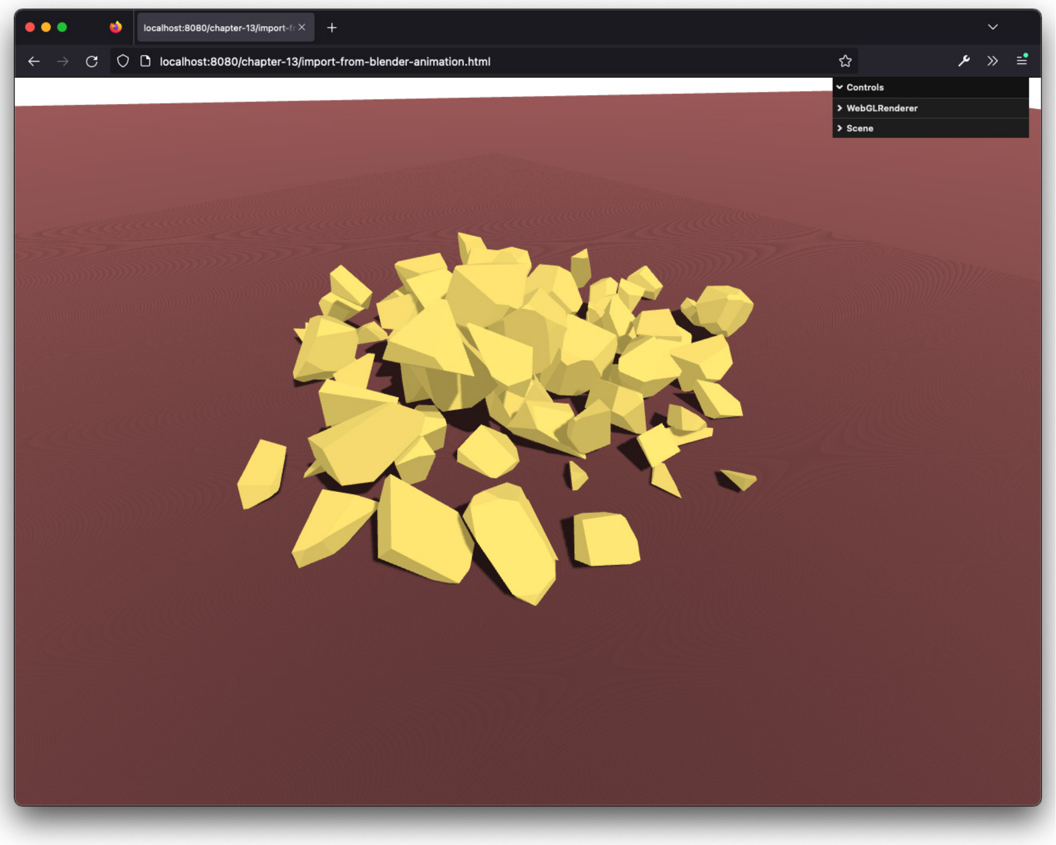

localhost:8080/chapter-13/import-f
localhost:8080/chapter-13/import-from-blender-animation.html
Controls
WebGLRenderer
Scene

Figure 13.18 – An exploded cube in Three.js

The same principle we've shown you here can be applied to different kinds of animations supported by Blender. The main thing to keep in mind is that Three.js won't understand physics engines used by Blender, or other advanced animation models. Therefore, when you export an animation, make sure you bake the animation so that you can use the standard Three.js tools to play back these keyframe-based animations.

For the next section, we're going to look a bit closer at how you can use Blender to bake different kinds of textures (maps) that you can then load into Three.js. We've already seen the results in action in *Chapter 10*, *Loading and Working with Textures*, but in this section, we'll show you how to use Blender to bake these maps.

Baking lightmaps and ambient occlusion maps in Blender

For this scenario, we're going to revisit the example from *Chapter 10*, where we used a lightmap from Blender. This lightmap provides good-looking lighting without having to calculate it in real time in Three.js. To do this with Blender, we're going to take the following steps:

1. Set up a simple scene in Blender with a couple of models.
2. Set up the lighting and the models in Blender.
3. Bake the lighting to textures in Blender.
4. Export the scene.
5. Render everything in Three.js.

In the following sections, we will discuss each step in detail.

Setting up a scene in Blender

For this example, we'll create a simple scene in which we'll bake in some lighting. Start a new project, delete the default cube by selecting it and hitting **x**, and do the same for the default light. Use **Add | Mesh | Plane** to add a simple 2D plane to the scene. Press *Tab* to go to **Edit Mode**, select three vertices, and extrude `e` and then `z` to extrude along the *z*-axis to get a simple shape like this:

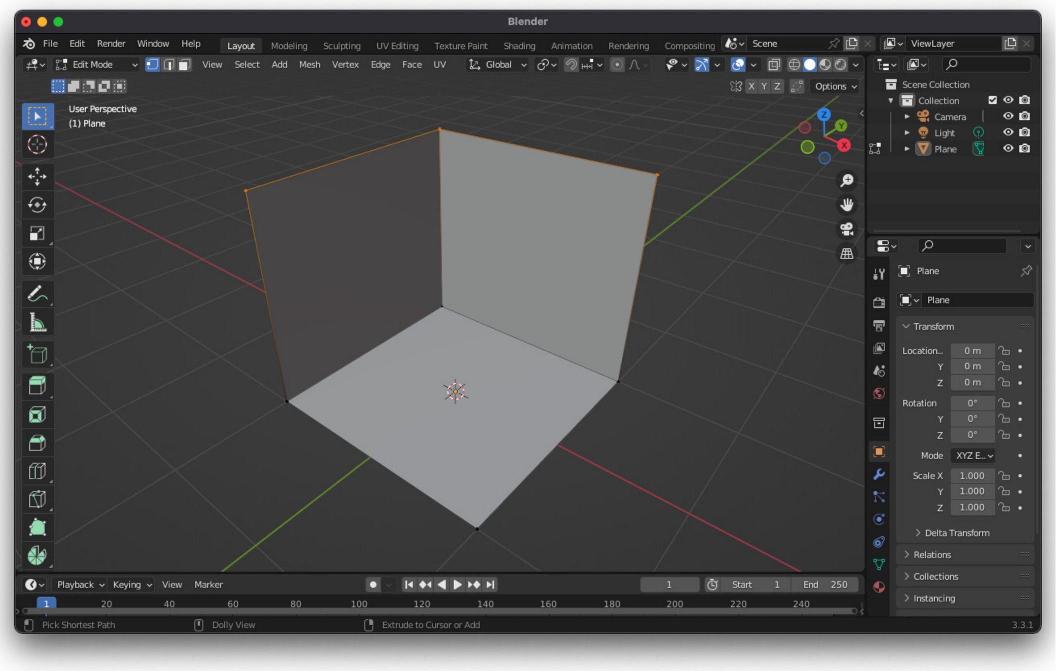
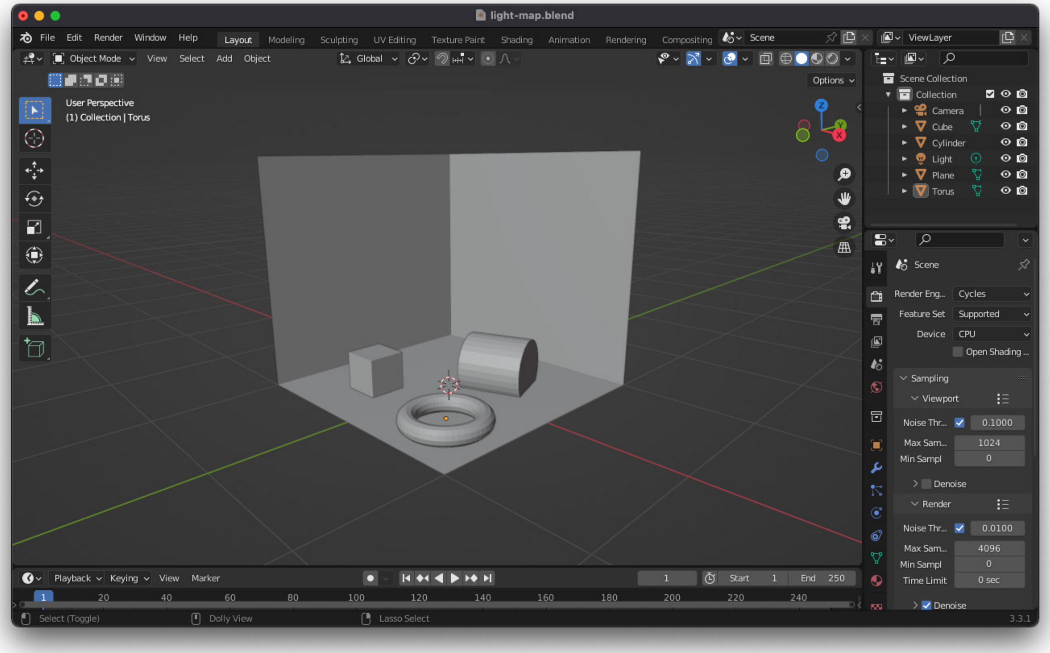

Blender
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing
Scene
ViewLayer
Edit Mode
View Select Add Mesh Vertex Edge Face UV
Global
Options
User Perspective
(1) Plane
Scene Collection
Collection
Camera
Light
Plane
Transform
Location... 0 m
Y 0 m
Z 0 m
Rotation 0°
Y 0°
Z 0°
Mode XYZ E...
Scale X 1.000
Y 1.000
Z 1.000
Delta Transform
Relations
Collections
Instancing
Playback Keying View Marker
Start 1 End 250
Pick Shortest Path
Dolly View
Extrude to Cursor or Add
3.3.1

Figure 13.19 – Creating a simple room structure

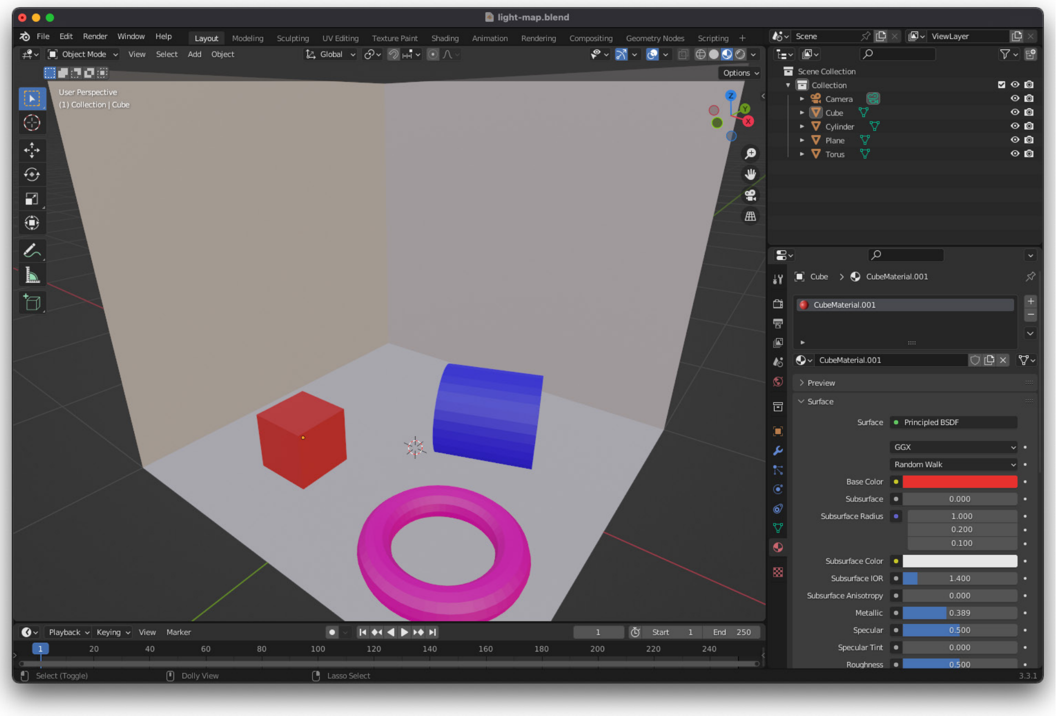
Once you've got this model, go back to **Object Mode** (using *Tab*), and place a couple of meshes in the room to get something looking similar to what is shown here:


light-map.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(1) Collection | Torus
Scene Collection
Collection
Camera
Cube
Cylinder
Light
Plane
Torus
Playback
Keying
View
Marker
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Scene
Render Eng... Cycles
Feature Set Supported
Device CPU
☐ Open Shading ...
Sampling
Viewport
☑ Noise Thr... 0.1000
Max Sam... 1024
Min Sampl 0
☐ Denoise
Render
☑ Noise Thr... 0.0100
Max Sam... 4096
Min Sampl 0
Time Limit 0 sec
☑ Denoise
1 Start 1 End 250

Figure 13.20 – A full room with some meshes

Nothing special at this point – just a simple room without any lighting. Before we move on to adding some lighting, change the colors of the objects a bit. Therefore, in Blender, go to **Material Properties**, create a new material for each mesh, and set a color. The result will look something similar to this:


light-map.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(1) Collection | Cube
Scene
ViewLayer
Scene Collection
Collection
Camera
Cube
Cylinder
Plane
Torus
Playback
Keying
View
Marker
Start 1
End 250
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Cube > CubeMaterial.001

CubeMaterial.001

Preview

Surface

Surface: Principled BSDF

GGX

Random Walk

Base Color: 🟥

Subsurface: 0.000

Subsurface Radius: 1.000, 0.200, 0.100

Subsurface Color: ⬜

Subsurface IOR: 1.400

Subsurface Anisotropy: 0.000

Metallic: 0.389

Specular: 0.500

Specular Tint: 0.000

Roughness: 0.500

Figure 13.21 – Adding colors to the different objects in the scene

Next, we'll add some nice lighting.

Adding lighting to the scene

For the lighting in this scene, we'll add nice HDRI-based lighting. With HDRI lighting, we don't have a single source of light but provide an image that'll be used as the source of light for the scene. For this example, we've downloaded an HDRI image from here: `https://polyhaven.com/a/thatch_chapel`.

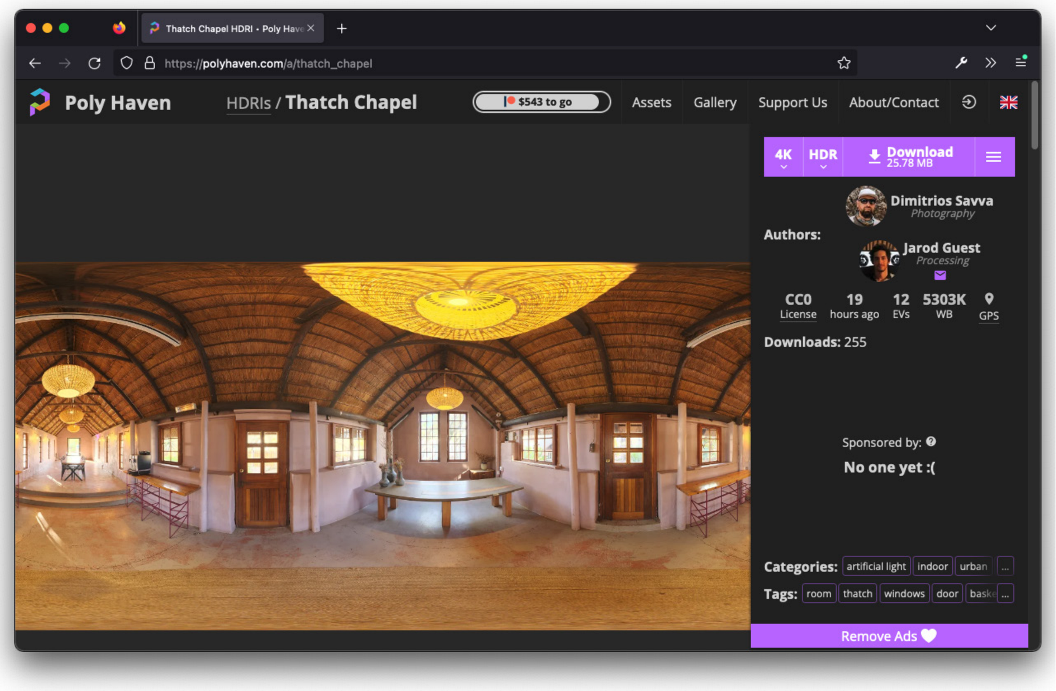

Thatch Chapel HDRI • Poly Have
https://polyhaven.com/a/thatch_chapel
Poly Haven
HDRIs / Thatch Chapel
$543 to go
Assets
Gallery
Support Us
About/Contact
4K
HDR
Download 25.78 MB
Authors:
Dimitrios Savva
Photography
Jarod Guest
Processing
CC0 License
19 hours ago
12 EVs
5303K WB
GPS
Downloads: 255
Sponsored by:
No one yet :(
Categories: artificial light indoor urban ...
Tags: room thatch windows door bask ...
Remove Ads

Figure 13.22 – Downloading an HDRI from Poly Haven

After downloading, we have a large image file that we can use in Blender. For this, open up the **World** tab from the **Properties Editor** panel, select the **Surface** dropdown, and select **Background**. Below this, you'll find the **Color** option, click this, and select **Environment Texture**:

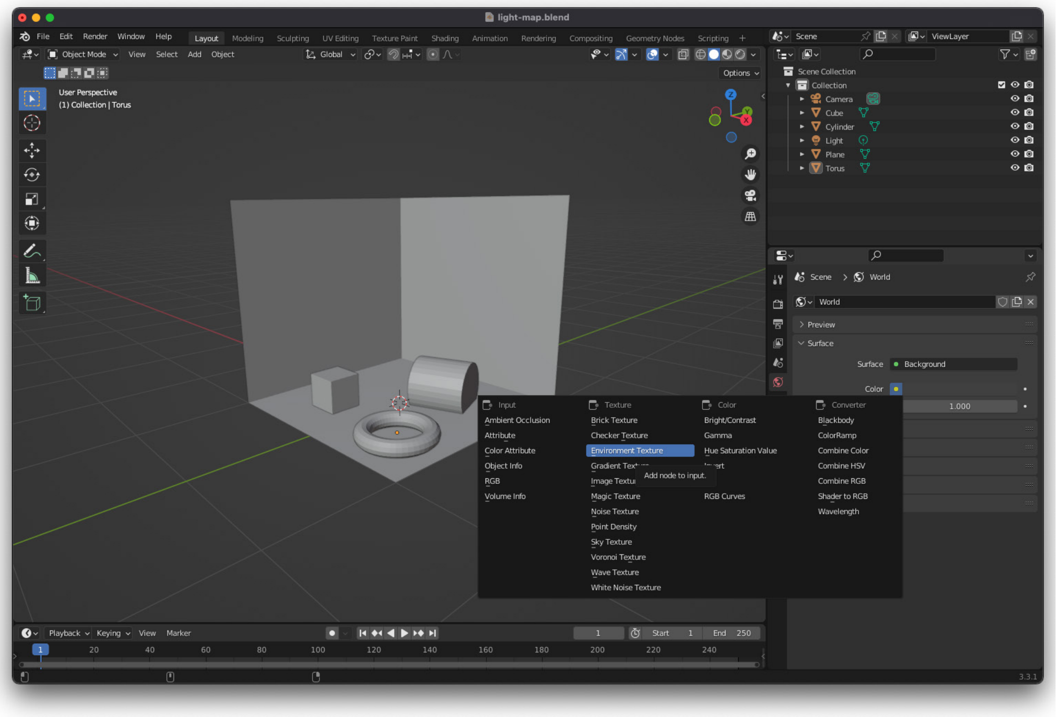
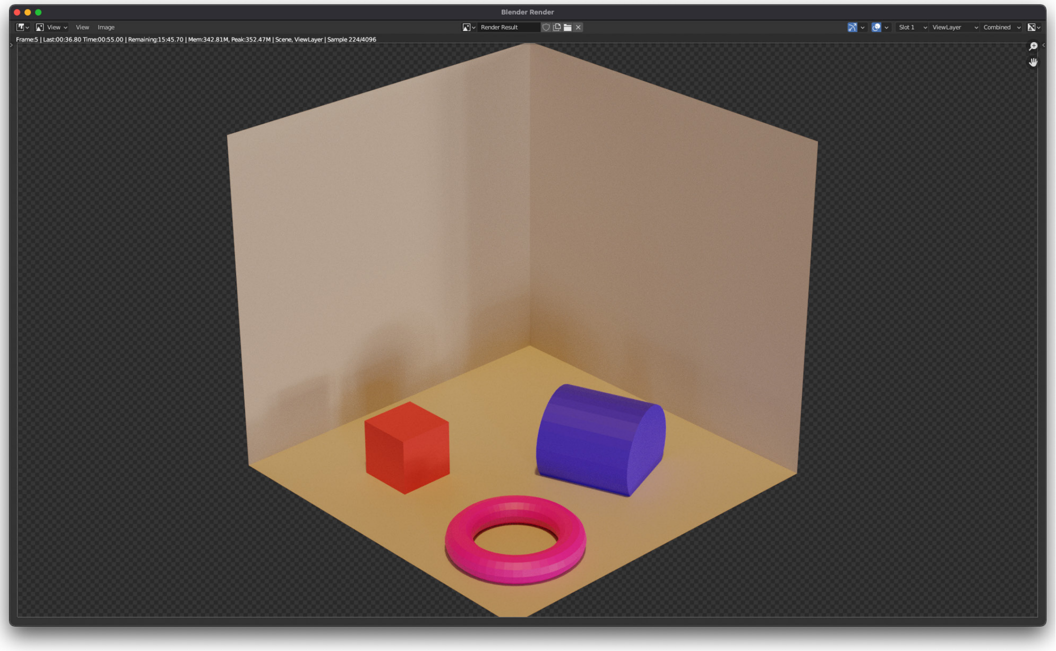

light-map.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Scene
ViewLayer
Object Mode
View
Select
Add
Object
Global
Options
User Perspective
(1) Collection | Torus
Scene Collection
Collection
Camera
Cube
Cylinder
Light
Plane
Torus
Scene
World
World
Preview
Surface
Surface
Background
Color
1.000
Input
Ambient Occlusion
Attribute
Color Attribute
Object Info
RGB
Volume Info
Texture
Brick Texture
Checker Texture
Environment Texture
Gradient Texture
Image Texture
Magic Texture
Noise Texture
Point Density
Sky Texture
Voronoi Texture
Wave Texture
White Noise Texture
Add node to input.
Color
Bright/Contrast
Gamma
Hue Saturation Value
Invert
RGB Curves
Converter
Blackbody
ColorRamp
Combine Color
Combine HSV
Combine RGB
Shader to RGB
Wavelength
Playback
Keying
View
Marker
1
Start
1
End
250
1
20
40
60
80
100
120
140
160
180
200
220
240
3.3.1

Figure 13.23 – Adding an environment texture to the world

Next, click on **Open**, browse to where you downloaded the image, and select that location. At this point, we can just render the scene and see what the HDRI map provides as lighting:



```
Frame:5 | Last:00:36.80 Time:00:55.00 | Remaining:15:45.70 | Mem:342.81M, Peak:352.47M | Scene, ViewLayer | Sample 224/4096
```

Figure 13.24 – Rendering the scene to check out the HDRI lighting

As you can see here, the scene already looks quite nice without us having to position separate lights. We now have some nice soft shadows on the walls, the objects seem to be lit from multiple angles, and the objects look nice. To use the information from the lights as a static lightmap, we need to bake the lighting onto a texture and map that texture to the objects in Three.js.

Baking the light textures

To bake the lights, first, we have to create a texture to hold this information. Select the cube (or any of the other objects you want to bake the lighting for). Go to the **Shading** view, and in **Node Editor** at the bottom of the screen, add a new **Image Texture** item: **Add | Texture | Image Texture**. The default values should be good to use:

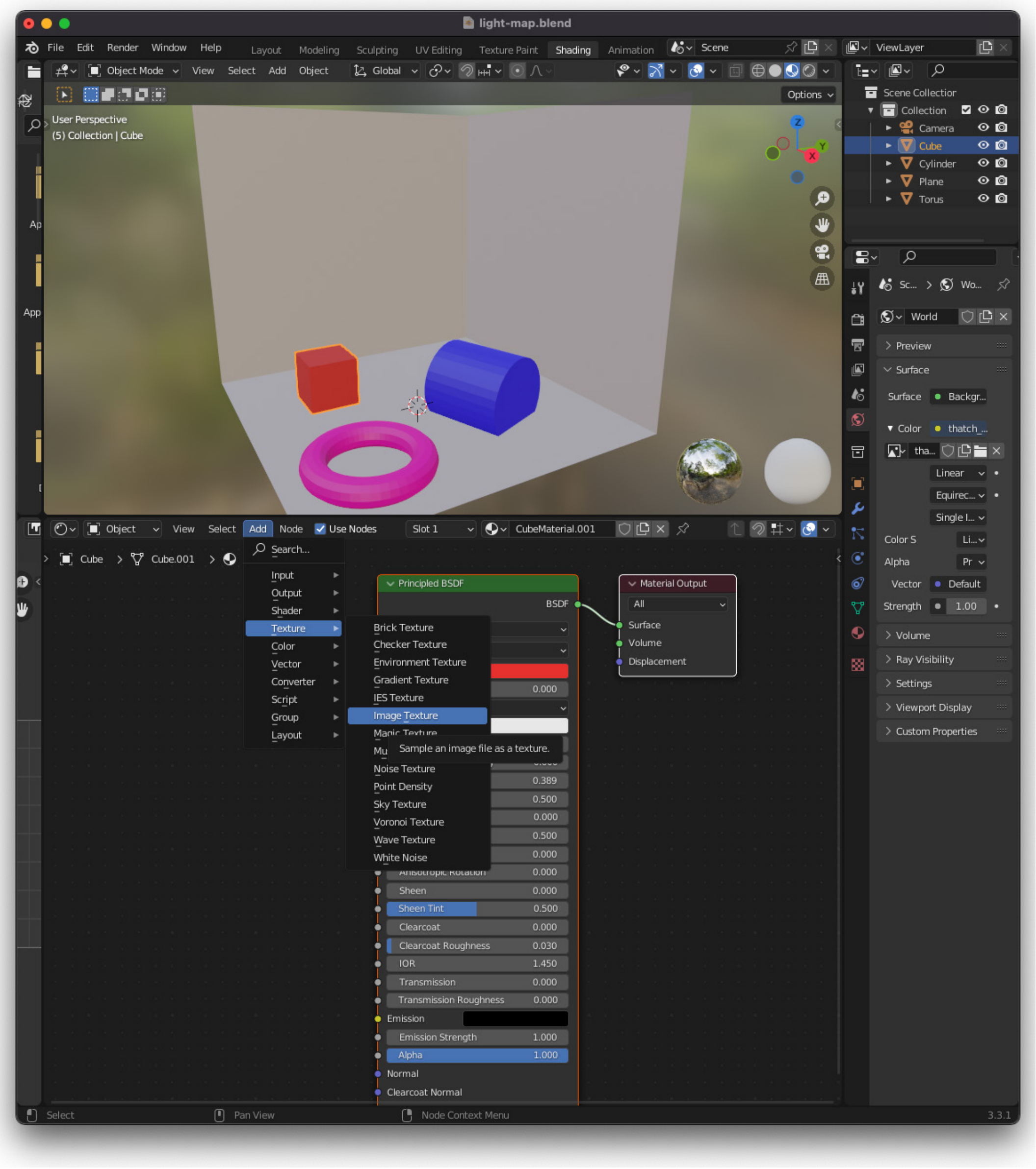
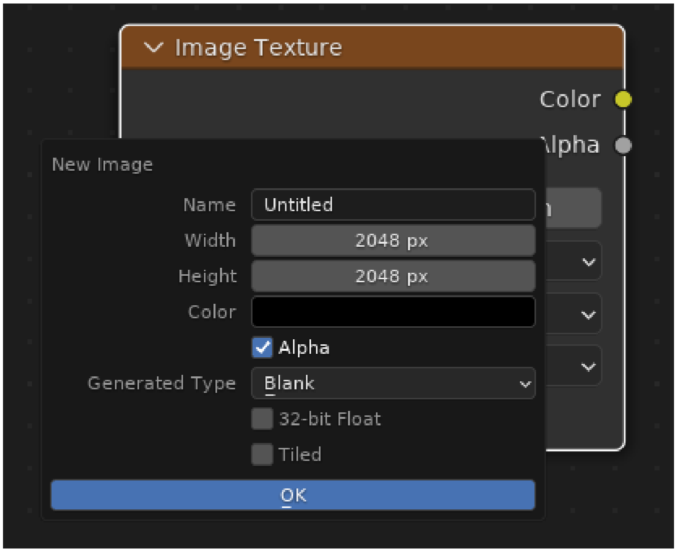

light-map.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Scene
ViewLayer
Object Mode
Global
Options
User Perspective
(5) Collection | Cube
Scene Collection
Collection
Camera
Cube
Cylinder
Plane
Torus
World
Preview
Surface
Surface
Backgr...
Color
thatch_...
Linear
Equirec...
Single I...
Color S
Alpha
Vector
Default
Strength
1.00
Volume
Ray Visibility
Settings
Viewport Display
Custom Properties
Object
View
Select
Add
Node
Use Nodes
Slot 1
CubeMaterial.001
Cube
Cube.001
Search...
Input
Output
Shader
Texture
Color
Vector
Converter
Script
Group
Layout
Brick Texture
Checker Texture
Environment Texture
Gradient Texture
IES Texture
Image Texture
Magic Texture
Sample an image file as a texture.
Noise Texture
Point Density
Sky Texture
Voronoi Texture
Wave Texture
White Noise
Principled BSDF
BSDF
Material Output
All
Surface
Volume
Displacement
Sheen
Sheen Tint
Clearcoat
Clearcoat Roughness
IOR
Transmission
Transmission Roughness
Emission
Emission Strength
Alpha
Normal
Clearcoat Normal
Select
Pan View
Node Context Menu
3.3.1

Figure 13.25 – Adding a texture image to hold the baked lightmap

Next, click on the **New** button of the node you just added, and select the size and name of the texture:



New Image

Name: Untitled

Width: 2048 px

Height: 2048 px

Color:

☑ Alpha

Generated Type: Blank

☐ 32-bit Float

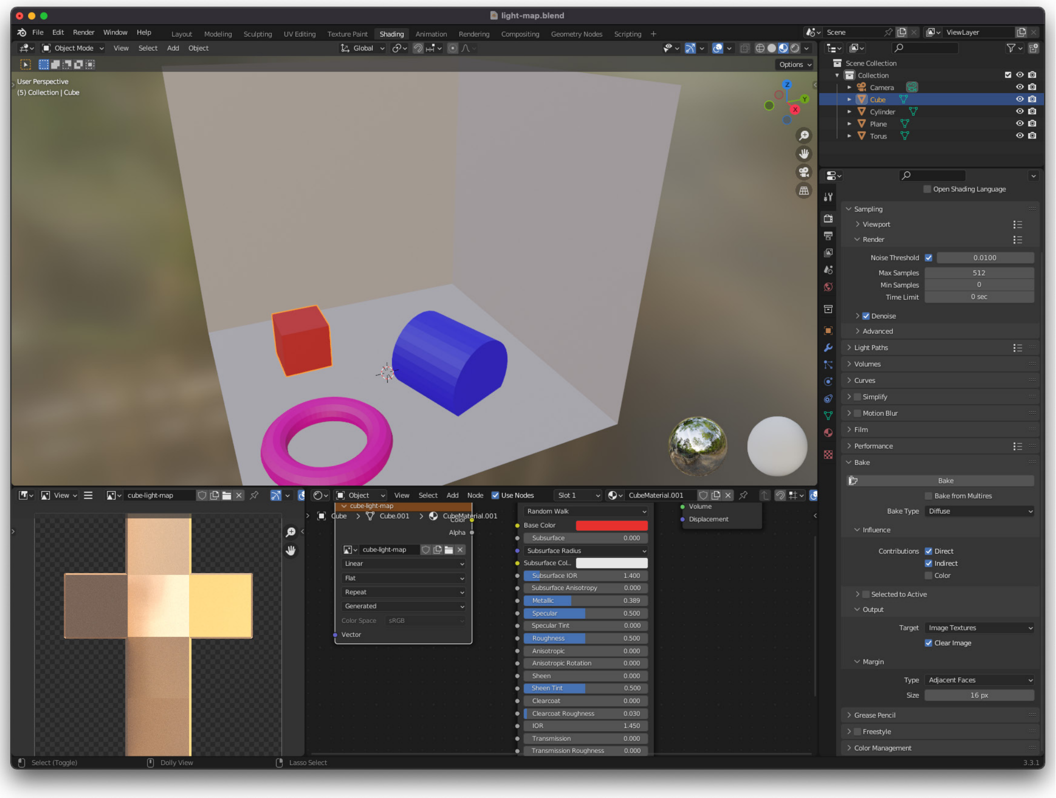
☐ Tiled

Figure 13.26 – Adding a new image to be used with the texture image

Now, go to the **Render** tab of the **Properties Editor** panel and set the following properties:

- **Render Engine**: **Cycles**.
- **Sampling | Render**: Set **Max Samples** to `512` or rendering the lightmap will take a very long time.
- In the **Bake** menu, select **Diffuse** from the **Bake Type** menu, and in the **Influence** section, select **Direct** and **Indirect**. This will just render the influence of our environment lighting.

Now, you can click **Bake** and Blender will render the lightmap for the selected object to the texture:



light-map.blend
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes Scripting
Object Mode View Select Add Object
Global
Options
User Perspective
(5) Collection | Cube
Scene
ViewLayer
Scene Collection
Collection
Camera
Cube
Cylinder
Plane
Torus
cube-light-map
Object View Select Add Node Use Nodes Slot 1 CubeMaterial.001
Cube Cube.001 CubeMaterial.001
Color
Alpha
Linear
Flat
Repeat
Generated
Color Space sRGB
Vector
Volume
Displacement
Select (Toggle)
Dolly View
Lasso Select
3.3.1

Random Walk
Base Color
Subsurface 0.000
Subsurface Radius
Subsurface Col..
Subsurface IOR 1.400
Subsurface Anisotropy 0.000
Metallic 0.389
Specular 0.500
Specular Tint 0.000
Roughness 0.500
Anisotropic 0.000
Anisotropic Rotation 0.000
Sheen 0.000
Sheen Tint 0.500
Clearcoat 0.000
Clearcoat Roughness 0.030
IOR 1.450
Transmission 0.000
Transmission Roughness 0.000

Open Shading Language
Sampling
Viewport
Render
Noise Threshold ☑ 0.0100
Max Samples 512
Min Samples 0
Time Limit 0 sec
☑ Denoise
Advanced
Light Paths
Volumes
Curves
☐ Simplify
☐ Motion Blur
Film
Performance
Bake
Bake
☐ Bake from Multires
Bake Type Diffuse
Influence
Contributions ☑ Direct
☑ Indirect
☐ Color
☐ Selected to Active
Output
Target Image Textures
☑ Clear Image
Margin
Type Adjacent Faces
Size 16 px
Grease Pencil
☐ Freestyle
Color Management

Figure 13.27 – Rendered lightmap for the cube

And that's it. As you can see in the image viewer on the bottom left, we've now got a nice-looking rendered lightmap for our cube. You can export this image as a standalone texture by clicking on the hamburger menu in the image viewer:

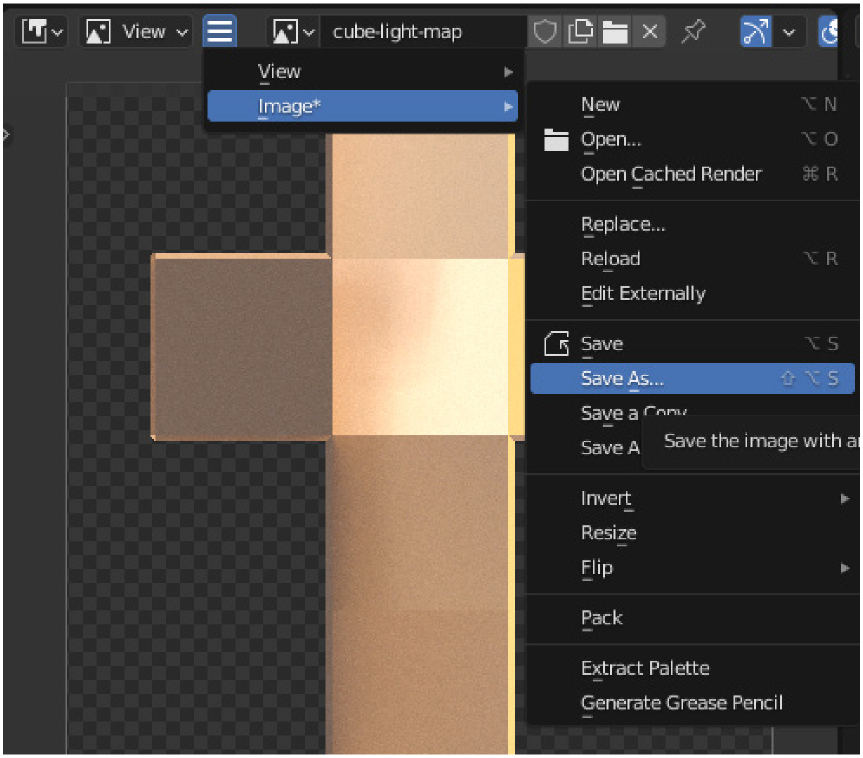
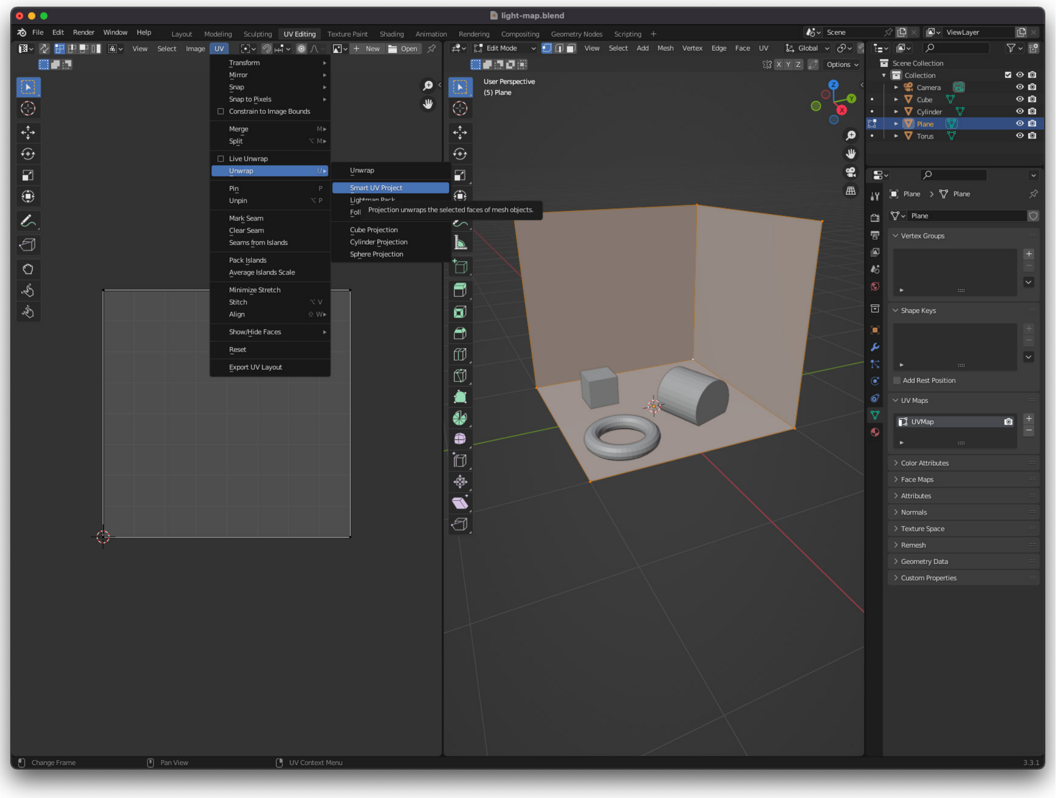

View
cube-light-map
View
Image*
New ⌥ N
Open... ⌥ O
Open Cached Render ⌘ R
Replace...
Reload ⌥ R
Edit Externally
Save ⌥ S
Save As... ⇧ ⌥ S
Save a Copy
Save A
Save the image with a
Invert
Resize
Flip
Pack
Extract Palette
Generate Grease Pencil

Figure 13.28 – Exporting the lightmap to an external file

You can now repeat this for the other meshes. Before doing this for the box though, we quickly need to fix the UV mapping. We need to do this since we extruded a couple of vertices to make the room-like structure, and Blender needs to know how to map them correctly. Without going into too much detail here, we can have Blender make a proposal on how to create the UV mapping. Click on the **UV Editing** menu at the top, select **Plane**, go to **Edit Mode**, and from the **UV** menu, select **UV | Unwrap | Smart Unwrap**:


light-map.blend
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Scene
ViewLayer
View
Select
Image
UV
New
Open
Transform
Mirror
Snap
Snap to Pixels
Constrain to Image Bounds
Merge
Split
Live Unwrap
Unwrap
Pin
Unpin
Mark Seam
Clear Seam
Seams from Islands
Pack Islands
Average Islands Scale
Minimize Stretch
Stitch
Align
Show/Hide Faces
Reset
Export UV Layout
Unwrap
Smart UV Project
Lightmap Pack
Follow
Cube Projection
Cylinder Projection
Sphere Projection
Edit Mode
View
Select
Add
Mesh
Vertex
Edge
Face
UV
Global
Options
User Perspective
(5) Plane
Scene Collection
Collection
Camera
Cube
Cylinder
Plane
Torus
Plane
Plane
Plane
Vertex Groups
Shape Keys
Add Rest Position
UV Maps
UVMap
Color Attributes
Face Maps
Attributes
Normals
Texture Space
Remesh
Geometry Data
Custom Properties
Change Frame
Pan View
UV Context Menu
3.3.1

Projection unwraps the selected faces of mesh objects.

Figure 13.29 – Fixing the UVs for the room mesh

This will make sure that a lightmap will be generated for all sides of the room. Now, repeat this for all the meshes, and you will have the lightmaps for this specific scene. Once you've exported all the lightmaps, we can export the scene itself and after that, render it using these created lightmaps in Three.js:

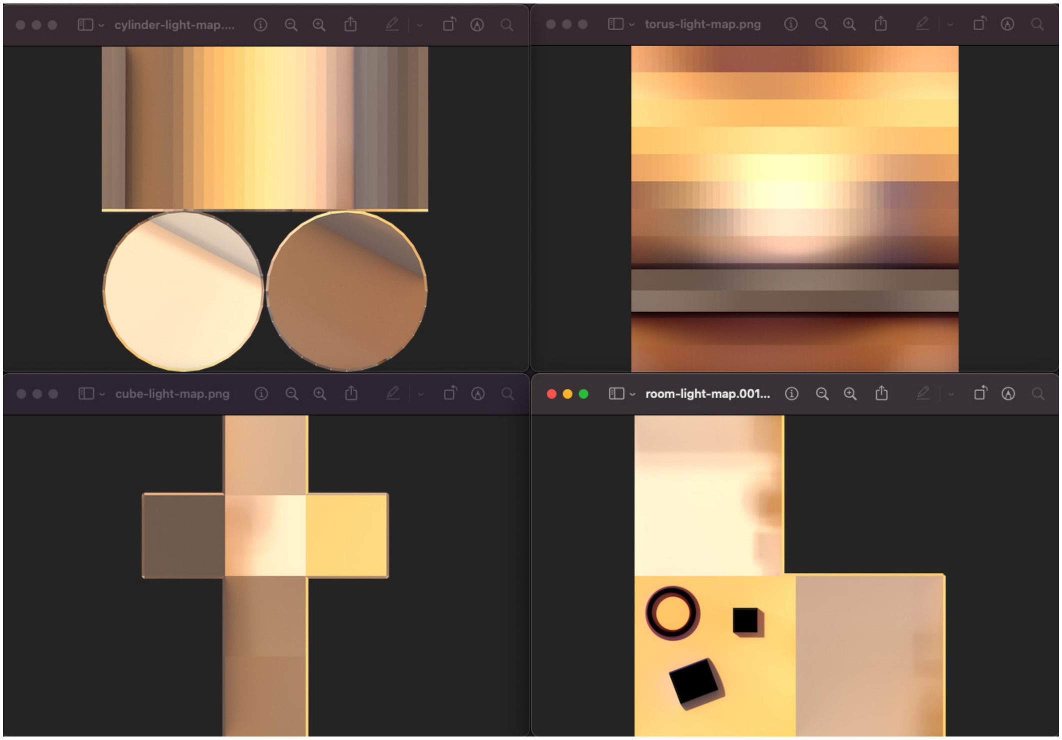


Figure 13.30 – All the created lightmaps

Now that we've baked all our maps, the next step is exporting everything from Blender and importing the scene and maps in Three.js.

Exporting the scene and importing it into Blender

We've already seen in the *Exporting a static scene from Blender and importing it into Three.js* section how to export a scene from Blender to be used in Three.js, so we'll repeat these same steps. Click on **File | Export | glTF 2.0**. We can use the defaults and since we don't have an animation, we can disable the animation checkbox. After exporting it, we can import the scene into Three.js. If we don't apply the texture (and use our own default lights), the scene will look something like this:

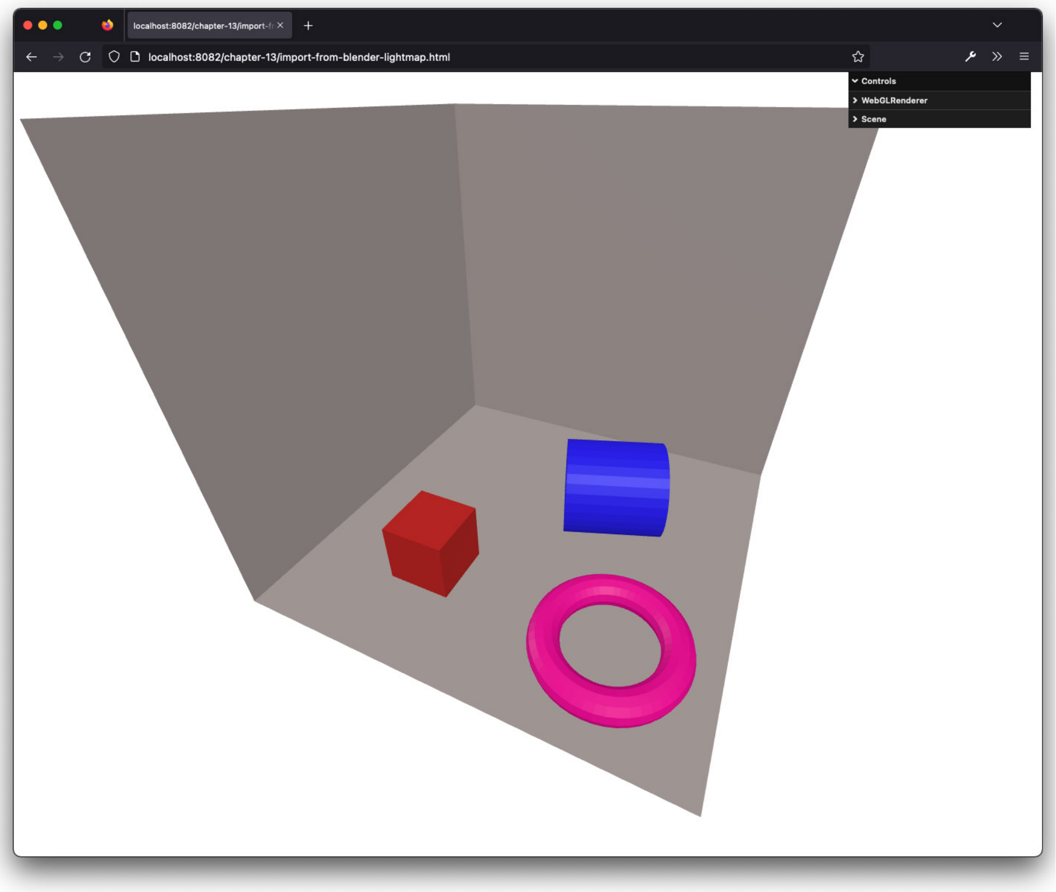

localhost:8082/chapter-13/import-f
localhost:8082/chapter-13/import-from-blender-lightmap.html
Controls
WebGLRenderer
Scene

Figure 13.31 – The Three.js scene rendered with default lights without the lightmaps

We've already seen how to load and apply a lightmap in *Chapter 10*. The following code fragment shows how to load the lightmap textures for all the lightmaps we've exported from Blender:

```
const cubeLightMap = new THREE.TextureLoader().load
  ('/assets/models/blender-lightmaps/cube-light-map.png')
const cylinderLightMap = new THREE.TextureLoader().load
('/assets/models/blender-lightmaps/cylinder-light-map.png')
const roomLightMap = new THREE.TextureLoader().load
  ('/assets/models/blender-lightmaps/room-light-map.png')
const torusLightMap = new THREE.TextureLoader().load
  ('/assets/models/blender-lightmaps/torus-light-map.png')
```

```
const addLightMap = (mesh, lightMap) => {
  const uv1 = mesh.geometry.getAttribute('uv')
  const uv2 = uv1.clone()
  mesh.geometry.setAttribute('uv2', uv2)
  mesh.material.lightMap = lightMap
  lightMap.flipY = false
}
const modelAsync = () => {
  const loader = new GLTFLoader()
  return loader.loadAsync('/assets/models/blender-
    lightmaps/light-map.glb').then((structure) => {
    const cubeMesh = structure.scene.
      getObjectByName('Cube')
    const cylinderMesh = structure.scene.
      getObjectByName('Cylinder')
    const torusMesh = structure.scene.
      getObjectByName('Torus')
    const roomMesh = structure.scene.
      getObjectByName('Plane')
    addLightMap(cubeMesh, cubeLightMap)
    addLightMap(cylinderMesh, cylinderLightMap)
    addLightMap(torusMesh, torusLightMap)
    addLightMap(roomMesh, roomLightMap)
    return structure.scene
  })
}
```

Now, when we look at the same scene (`import-from-blender-lightmap.html`), we have a scene with very nice lighting, even though we didn't provide any light sources ourselves and used the baked lights from Blender instead:

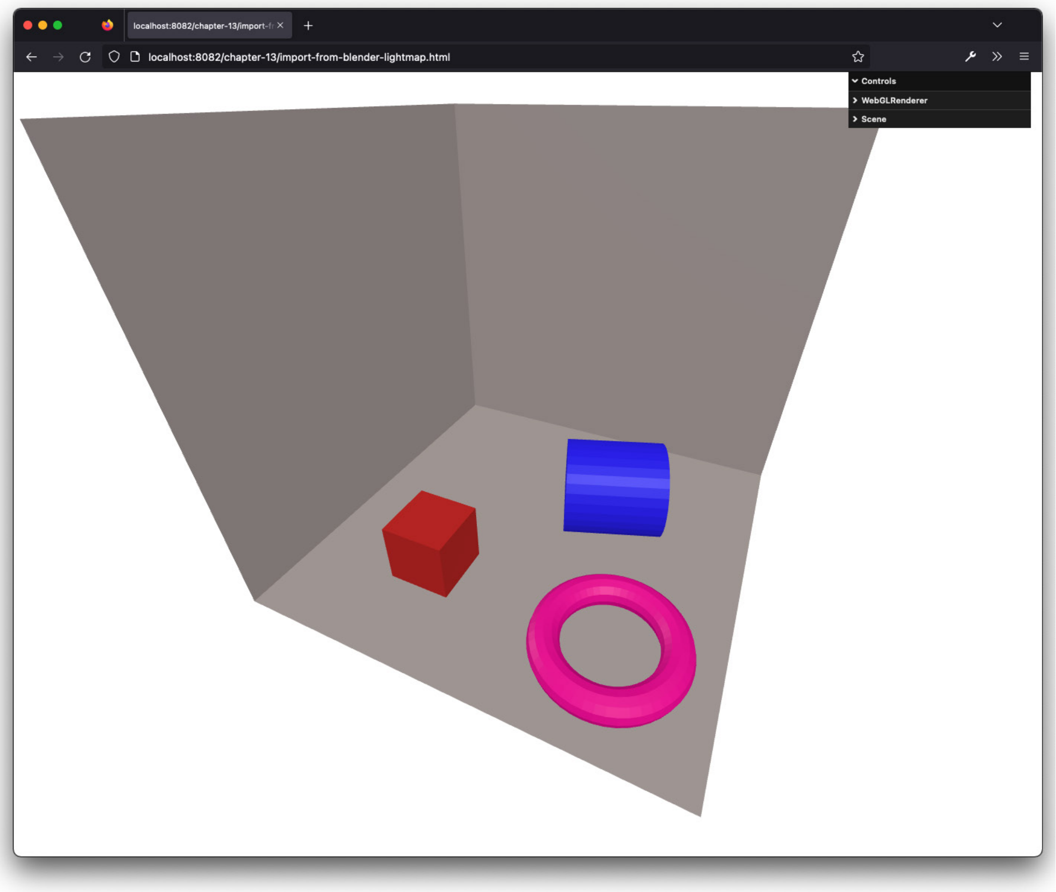

localhost:8082/chapter-13/import-f
localhost:8082/chapter-13/import-from-blender-lightmap.html
Controls
WebGLRenderer
Scene

Figure 13.32 – The same scene but with the baked lightmaps applied from Blender

If we export the lightmaps, we already implicitly get information about the shadows as well since, at those locations, there is, of course, less light. We can also get more detailed shadow maps from Blender. For instance, we can generate ambient occlusion maps so we don't have to create those at runtime.

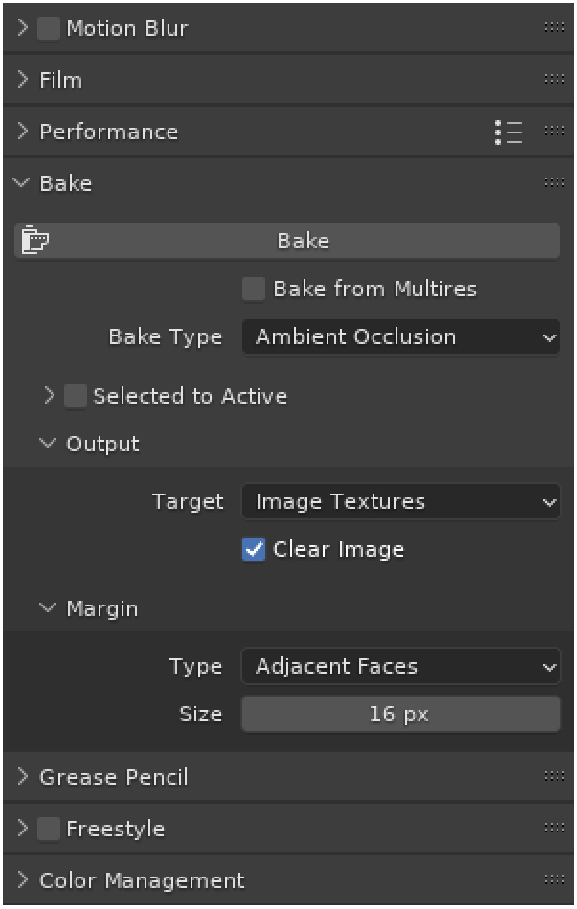
Baking an ambient occlusion map in Blender

If we go back to the scene we already have, we can also bake an ambient occlusion map. The approach for this is the same as that used for baking a lightmap:

1. Set up a scene.
2. Add all the lights and objects that cast shadows.

3. Make sure you've got an empty **Image Texture** in **Shader Editor**, to which we can bake the shadows.
4. Select the relevant baking options and render the shadows to the image.

Since the first three steps are the same as those for lightmaps, we'll skip those and look at the render settings needed to render the shadow maps:


Motion Blur
Film
Performance
Bake
Bake
Selected to Active
Output
Margin
Grease Pencil
Freestyle
Color Management

☐ Bake from Multires

Bake Type: Ambient Occlusion

Target: Image Textures

☑ Clear Image

Type: Adjacent Faces

Size: 16 px

Figure 13.33 – Render settings to bake an ambient occlusion map

As you can see, you only have to change the dropdown for **Bake Type** to **Ambient Occlusion**. Now, you can select the mesh for which you want to bake these shadows and click on the **Bake** button. For the room mesh, the result looks like this:

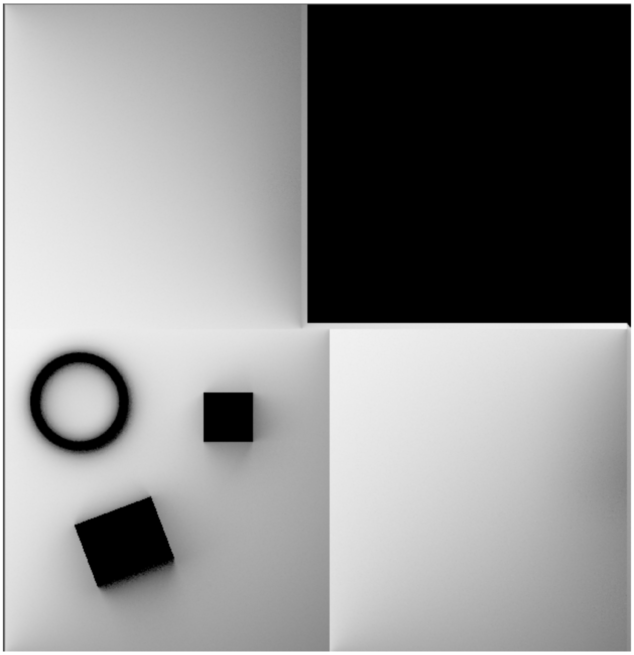


Figure 13.34 – An ambient occlusion map as a texture

Blender provides a number of other bake types that you can use to get good-looking textures (especially for the static parts of your scene), which can greatly improve the rendering performance.

There is one more subject we're going to look at for this section on Blender, and that is how to use Blender to change the UV mapping of a texture.

Custom UV modeling in Blender

In this section, we're going to start with a new empty Blender scene, and we'll use the default cube to experiment with. To get a good overview of how UV mapping works, you can use something called a UV grid, which looks something like this:

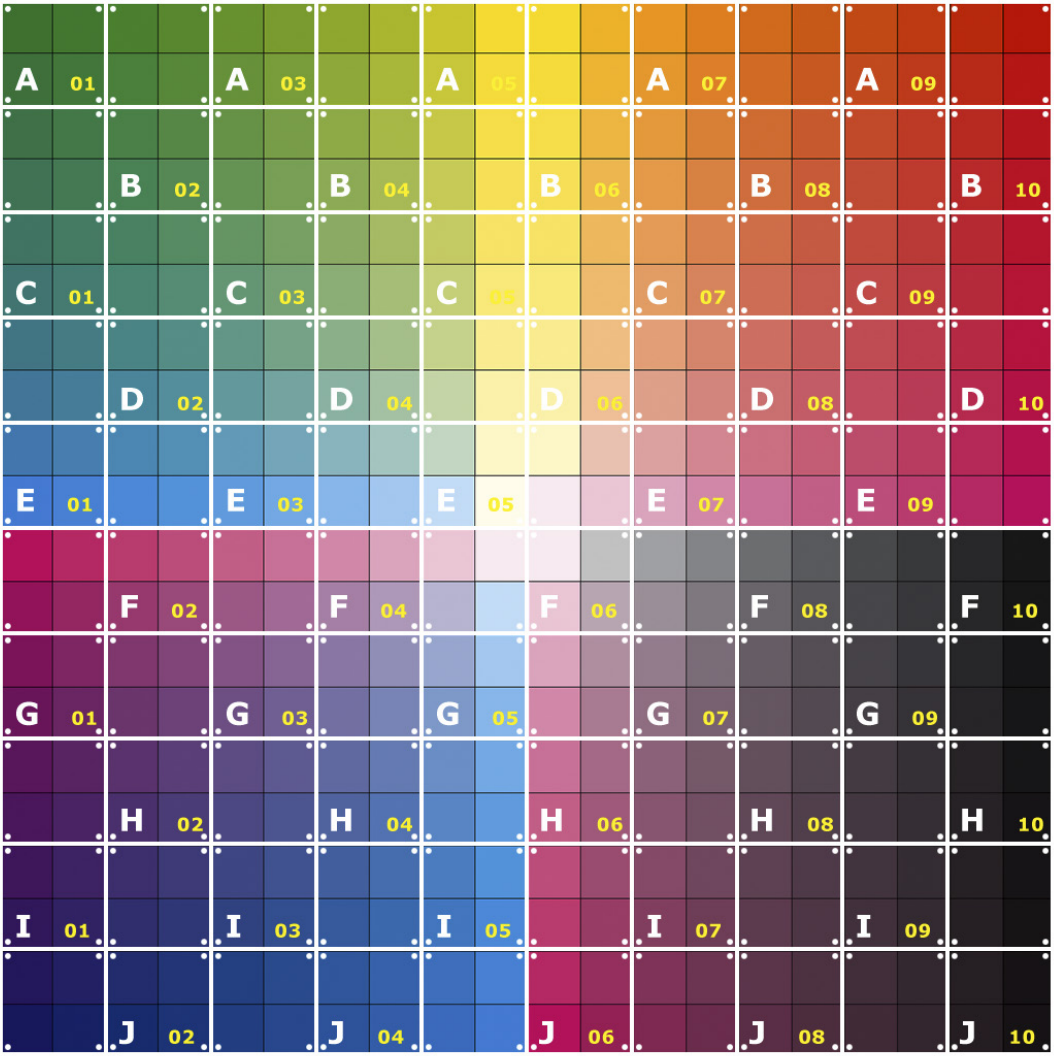
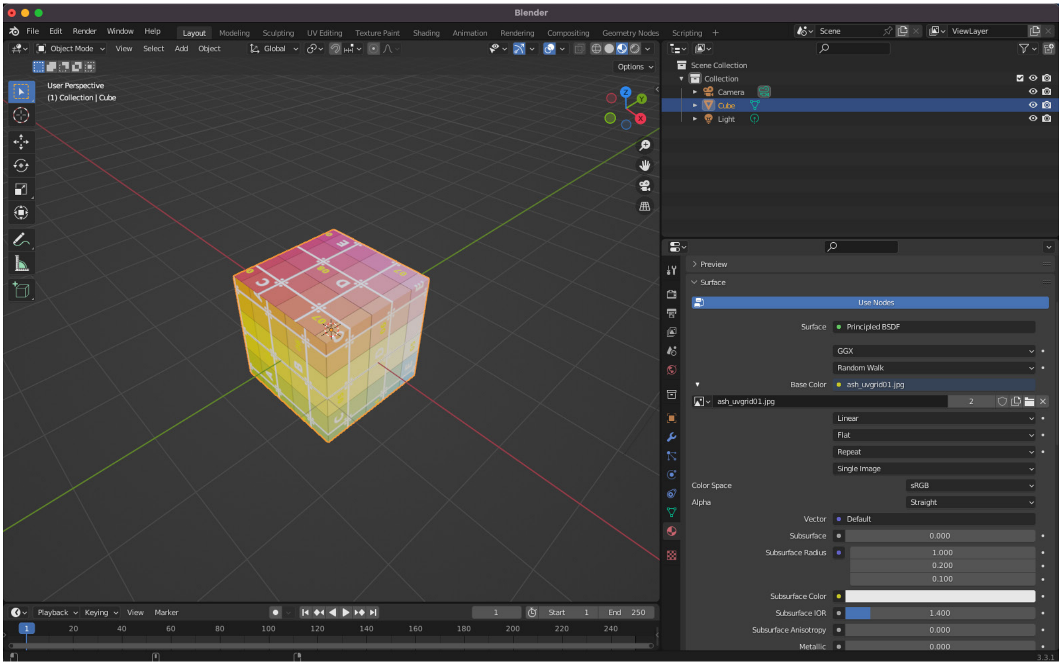

A 01 A 03 A 05 A 07 A 09
B 02 B 04 B 06 B 08 B 10
C 01 C 03 C 05 C 07 C 09
D 02 D 04 D 06 D 08 D 10
E 01 E 03 E 05 E 07 E 09
F 02 F 04 F 06 F 08 F 10
G 01 G 03 G 05 G 07 G 09
H 02 H 04 H 06 H 08 H 10
I 01 I 03 I 05 I 07 I 09
J 02 J 04 J 06 J 08 J 10

Figure 13.35 – A sample UV texture

When you apply this as the texture of the default cube, you'll see how the various vertices of a mesh map to a specific location on the texture. To use this, the first thing we need to do is define this texture. You can easily do this from **Material Properties** in the **Properties** view on the right of the screen. Click on the yellow dot before the **Base Color** property and select **Image Texture**. This allows you to browse for an image to use as texture:


Blender
File
Edit
Render
Window
Help
Layout
Modeling
Sculpting
UV Editing
Texture Paint
Shading
Animation
Rendering
Compositing
Geometry Nodes
Scripting
Object Mode
Global
Options
User Perspective
(1) Collection | Cube
Scene Collection
Collection
Camera
Cube
Light
Preview
Surface
Use Nodes

Surface: Principled BSDF
GGX
Random Walk
Base Color: ash_uvgrid01.jpg
ash_uvgrid01.jpg 2
Linear
Flat
Repeat
Single Image
Color Space: sRGB
Alpha: Straight
Vector: Default
Subsurface: 0.000
Subsurface Radius: 1.000, 0.200, 0.100
Subsurface IOR: 1.400
Subsurface Anisotropy: 0.000
Metallic: 0.000

Figure 13.36 – Adding a texture to a mesh in Blender

You can already see in the main viewport how this texture is applied to the cube. If we export this mesh including the material into Three.js and render it, we will see exactly the same mapping because Three.js will use the UV mapping defined by Blender (`import-from-blender-uv-map-1.html`):

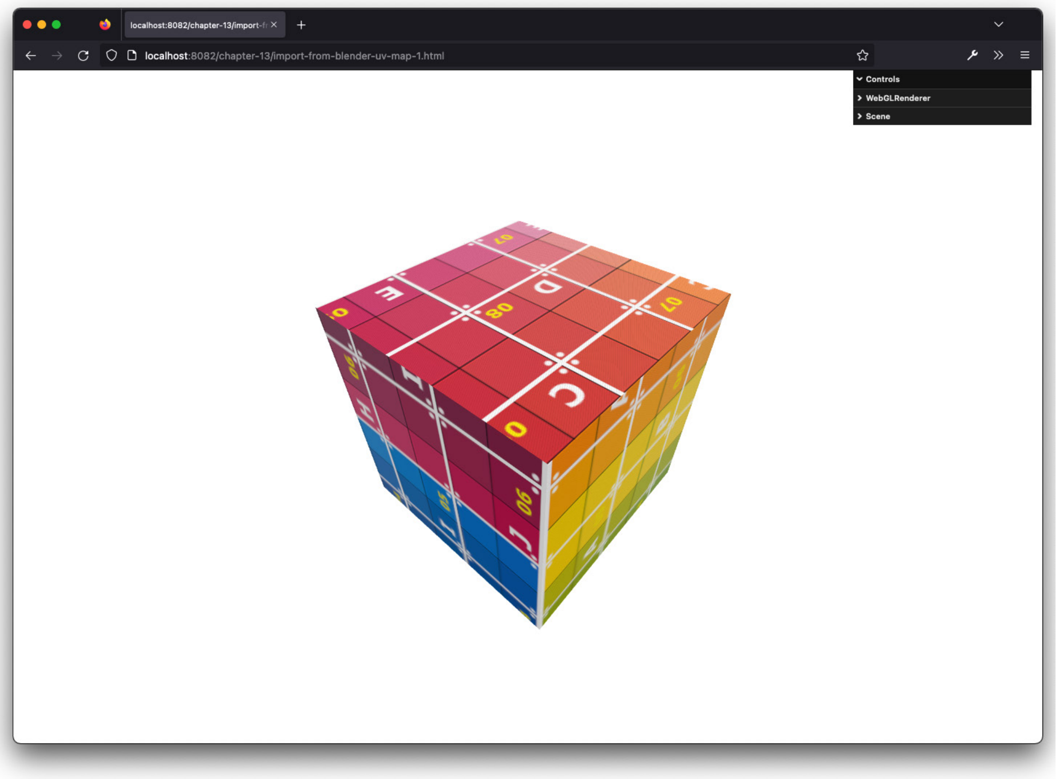

localhost:8082/chapter-13/import-f
localhost:8082/chapter-13/import-from-blender-uv-map-1.html
Controls
WebGLRenderer
Scene

Figure 13.37 – A box with a UV grid rendered in Three.js

Now, let's switch back to Blender, and open up the **UV Editing** tab. Go to **Edit Mode** (using the *Tab* key) on the right-hand side of the screen and select the four front-facing vertices. When you've selected these, you'll see the position of these four vertices on the left-hand side of the screen as well.

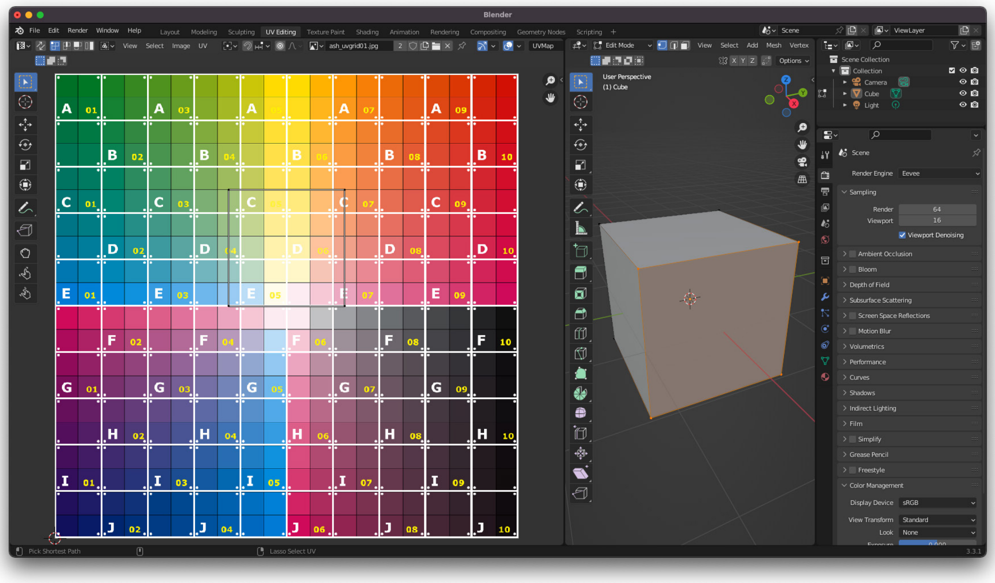


Figure 13.38 – The UV editor showing the mapping of the four pixels that make up the front face of the cube

In the UV editor, you can now grab (*g*) the vertices and move them to a different position on the texture. For instance, you can move them to the edges of the texture like this:

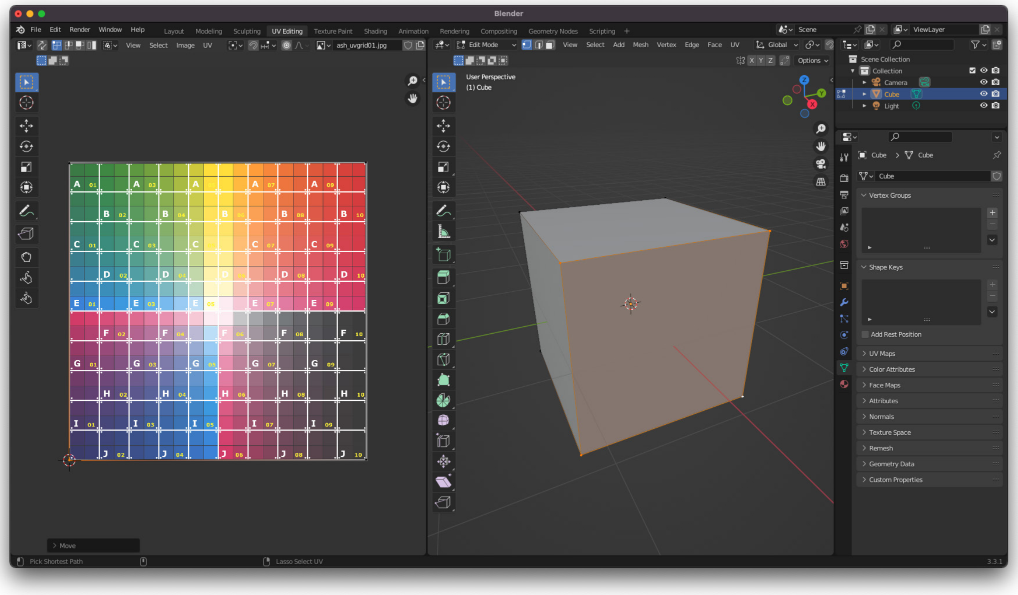
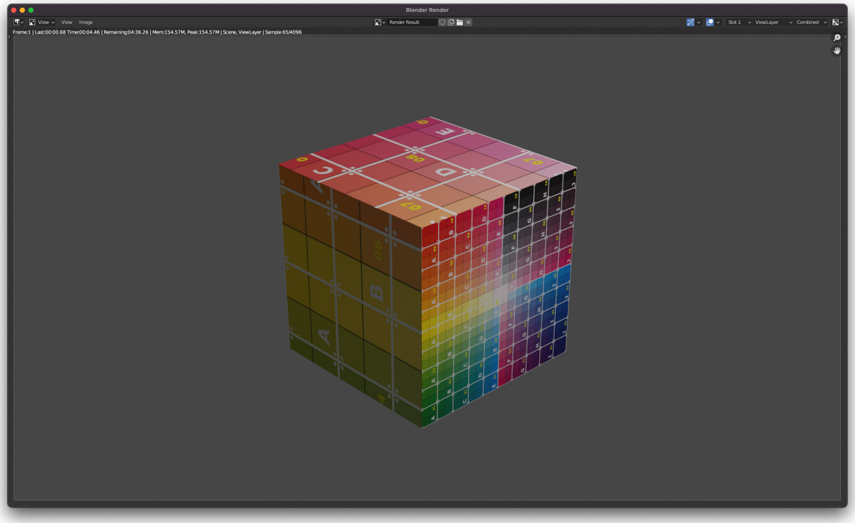


Figure 13.39 – One side mapped to the complete texture

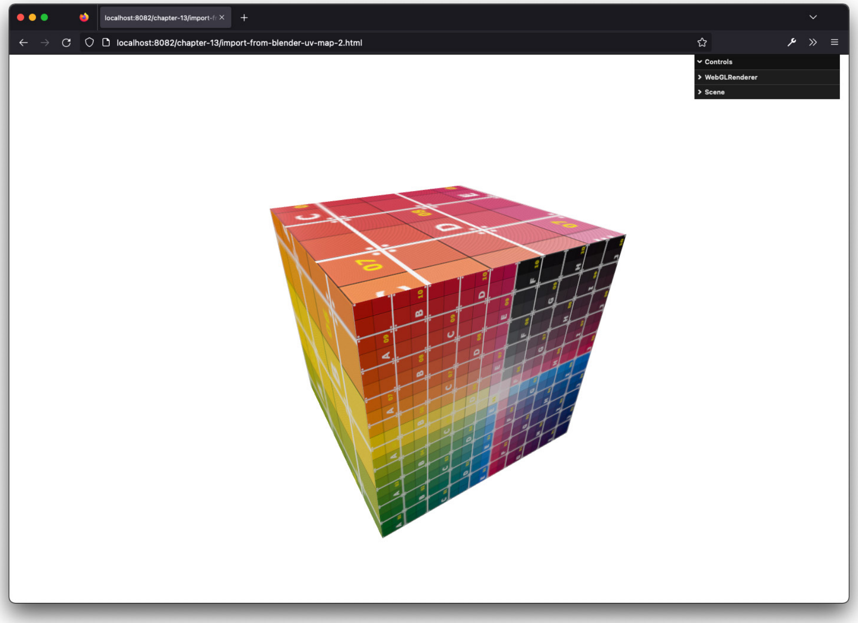
Moving the vertices will result in a cube that looks like this:


Blender Render
View
View
Image
Render Result

Frame:1 | Last:00:00.68 Time:00:04.46 | Remaining:04:36.26 | Mem:154.57M, Peak:154.57M | Scene, ViewLayer | Sample 65/4096

Figure 13.40 – The Blender render of the cube with custom UV mapping

And, of course, this is also directly shown in Three.js as well when we export and import this minimal model:


localhost:8082/chapter-13/import-f
Controls
WebGLRenderer
Scene

localhost:8082/chapter-13/import-from-blender-uv-map-2.html

Figure 13.41 – The Three.js view of the cube with custom UV mapping

Using this approach, it is very easy to define exactly which parts of your mesh should be mapped to which part of a texture.

Summary

In this chapter, we explored how you can work together with Blender and Three.js. We showed how you can use the glTF format as the standard format to exchange data between Three.js and Blender. This works great for meshes, animations, and most textures. However, for advanced texture properties, you will probably need some fine-tuning in either Three.js or Blender. We also showed how you can bake specific textures such as lightmaps and ambient occlusion maps in Blender and use them in Three.js. This allows you to render this information once in Blender, import it into Three.js, and create great shadows, lights, and ambient occlusion without the heavy calculations that Three.js would have to do normally. Note that this, of course, will only work for scenes where the lighting is static, and the geometries and meshes don't move around or change shape. Often, you can use this for the static parts of your scene. Finally, we looked a bit at how UV mapping works, where vertices are mapped to a position on a texture, and how you can use Blender to play around with this mapping. Once again, by using glTF as the exchange format, all the information from Blender can be easily used in Three.js.

We're now reaching the end of this book. In the last chapter, we're going to look at two additional subjects – how can you use Three.js together with React.js, and we'll have a quick look at Three.js's support for VR and AR.

14

Three.js Together with React, TypeScript, and Web-XR

In this final chapter, we'll dive into two additional topics. First, we'll look at how you can combine Three.js with TypeScript and React. The second part of this chapter will show some examples of how you can integrate your 3D scenes with Web-XR. With Web-XR, you can enhance your scenes to work with VR and AR technologies.

More specifically, we'll show you the following examples:

- **Using Three.js with TypeScript**: For the first example, we'll show you how you can create a simple project that combines Three.js and TypeScript. We'll create a very simple application, much like the samples we've already seen in the previous chapters, and show you how you can use TypeScript with Three.js to create your scenes.
- **Using Three.js and React with TypeScript**: React is a very popular framework for web development and is often used together with TypeScript. For this section, we'll create a simple Three.js project, which uses React.js together with TypeScript.
- **Using Three.js and React with Three.js fibers**: Finally, we'll look at `React-three-fiber`. With this library, we can configure Three.js declaratively using a set of React components. This library provides great integration between React and Three.js and makes working with Three.js in a React application straightforward.
- **Three.js and VR**: This section will show you how you can view your 3D scene in VR.
- **Three.js and AR**: This section will explain how you can create a simple 3D scene where you can add Three.js meshes.

Let's start with the first example of this chapter and integrate Three.js with TypeScript.

Using Three.js with TypeScript

TypeScript provides a typed language that transpiles to JavaScript. This means that you can use it to create your site, and it'll run just like normal JavaScript in the browser. There are many different ways of setting up a TypeScript project, but the easiest one is provided by Vite (`https://vitejs.dev/`). Vite provides an integrated build environment and can be seen a bit as an alternative to webpack (which we use for the normal chapter samples).

The first thing we need to do is create a new Vite project. You can do these steps yourself, or you can look in the `three-ts` folder and just run `yarn install` there to skip this setup. To get an empty TypeScript project with Vite, all we have to do is run the following code in the console:

```
$ yarn create vite three-ts --template vanilla-ts
yarn create v1.22.17
warning package.json: No license field
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
[4/4] 🔨  Building fresh packages...
warning Your current version of Yarn is out of date. The latest
version is "1.22.19", while you're on "1.22.17".
info To upgrade, run the following command:
$ curl --compressed -o- -L https://yarnpkg.com/install.sh |
bash
success Installed "create-vite@3.2.1" with binaries:
      - create-vite
      - cva
[###############################################################
########] 70/70
Scaffolding project in /Users/jos/dev/git/personal/ltjs4-all/
three-ts...
```

Next change into the directory (`three-ts`) and run `yarn install`.

```
$ yarn install
yarn install v1.22.17
warning ../package.json: No license field
```

```
info No lockfile found.
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
[4/4] 🔨  Building fresh packages...
success Saved lockfile.
✨  Done in 3.31s.
```

At this point, we've got an empty Vite project, which you can start by running `yarn vite`.

```
$  three-ts git:(main) ✗ yarn vite
yarn run v1.22.17
warning ../package.json: No license field
$ /Users/jos/dev/git/personal/ltjs4-all/three-ts/node_modules/.
bin/vite
  VITE v3.2.3  ready in 193 ms
  ➜  Local:   http://127.0.0.1:5173/
  ➜  Network: use --host to expose
```

If you point your browser to `http://127.0.0.1:5173/`, you'll see the start page of Vite, and you'll have a configured TypeScript project in place:

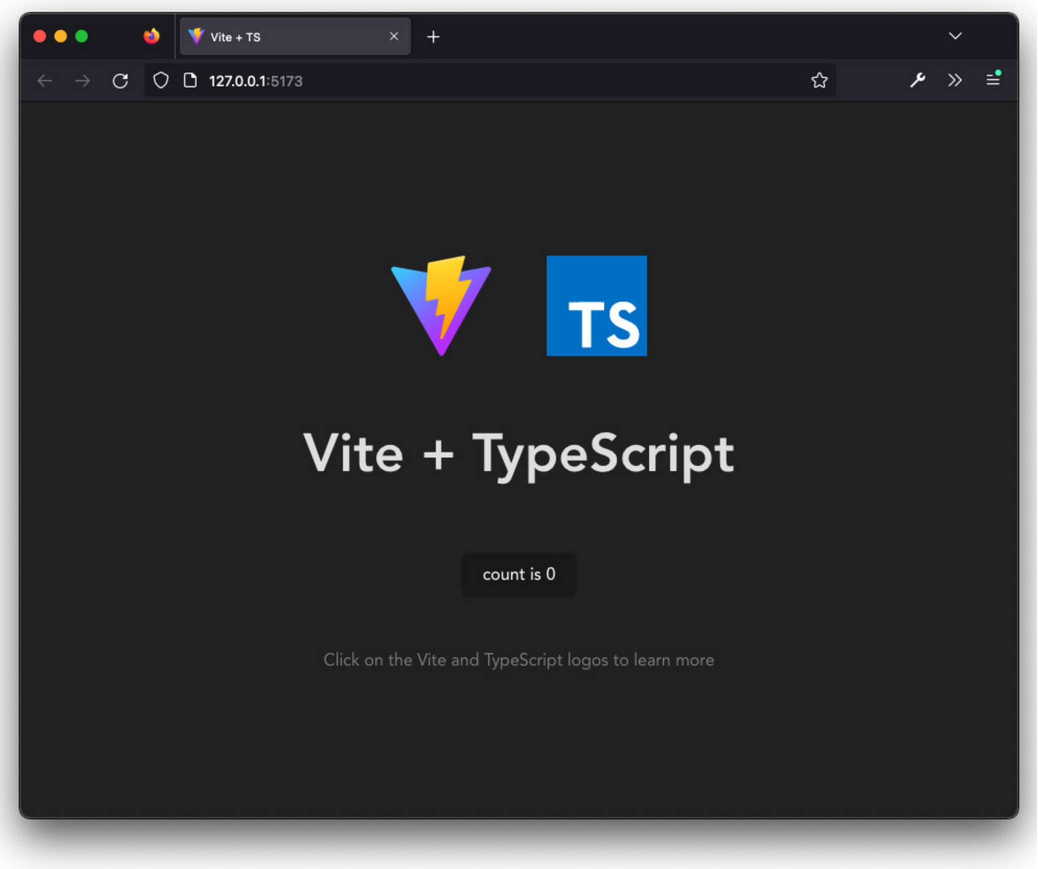

Vite + TS
127.0.0.1:5173
TS
Vite + TypeScript
count is 0
Click on the Vite and TypeScript logos to learn more

Figure 14.1 – Empty TypeScript project with Vite

Next, we must add the Three.js libraries, after which we can add some TypeScript to initialize Three.js. To add Three.js, we need to add the following two node modules:

```
$ yarn add three
$ yarn add -D @types/three
```

The first one adds the Three.js library, while the second one adds the `types` descriptions for the Three.js library. These `types` are used in the editor to get some nice code completion when working with Three.js and TypeScript in your IDE (for example, in Visual Studio Code). At this point, we're ready to add Three.js to this project and start developing Three.js applications using TypeScript. To add TypeScript, the first thing we need to do is take a quick look at how the application is initialized. For this, you can look at the `public.html` file, which looks like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width,
      initial-scale=1.0" />
    <title>Vite + TS</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

In the preceding code, as you can see in the last `script` line, this HTML page loads the `src/main/ts` file. Open this file and change its content to this:

```
import './style.css'
import { initThreeJsScene } from './threeCanvas'
const mainElement = document.querySelector
  <HTMLDivElement>('#app')
if (mainElement) {
  initThreeJsScene(mainElement)
}
```

The code here will try to find the main `#app` node. And if it finds the node, it'll pass that node to the `initThreeJsScene` function, which is defined in the `threeCanvas.ts` file. This file contains the code to initialize the Three.js scene:

```
import * as THREE from 'three'
import { OrbitControls } from 'three/examples/jsm/
  controls/OrbitControls'
export const width = 500
export const height = 500
export const initThreeJsScene = (node: HTMLDivElement) => {
  const scene = new THREE.Scene()
  const camera = new THREE.PerspectiveCamera(75, height /
    width, 0.1, 1000)
  const renderer = new THREE.WebGLRenderer()
  renderer.setClearColor(0xffffff)
  renderer.setSize(height, width)
  node.appendChild(renderer.domElement)
  camera.position.z = 5
  const geometry = new THREE.BoxGeometry()
  const material = new THREE.MeshNormalMaterial()
  const cube = new THREE.Mesh(geometry, material)
  const controls = new OrbitControls(camera, node)
  scene.add(cube)
  const animate = () => {
    controls.update()
    requestAnimationFrame(animate)
    cube.rotation.x += 0.01
    cube.rotation.y += 0.01
    renderer.render(scene, camera)
  }
  animate()
}
```

This will look familiar to the code from the first couple of chapters where we created an initial simple scene. The main change is that, here, we can use all the features provided by TypeScript. Vite will handle the transpiling to JavaScript, so you don't need to do anything else to see the results of this in your browser:

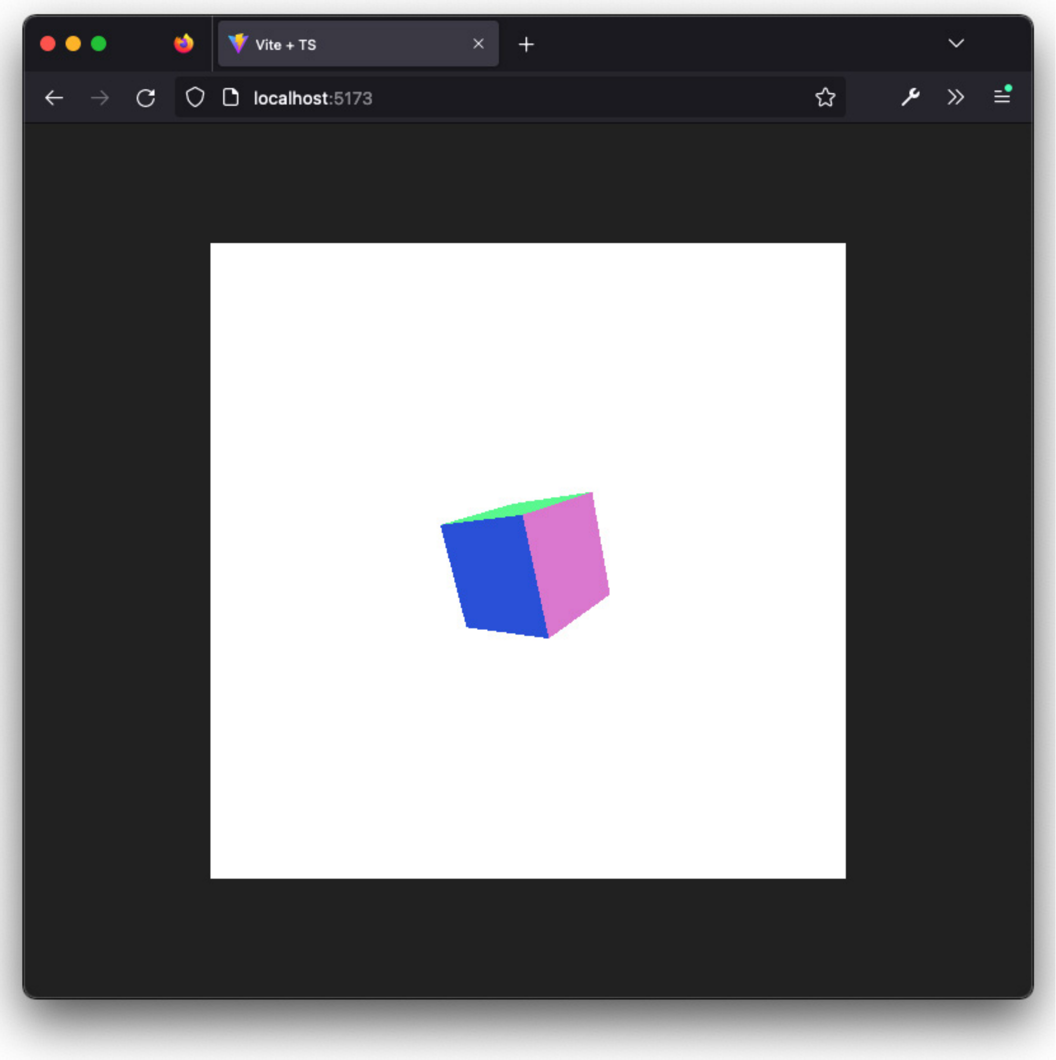


Figure 14.2 – Simple TypeScript project with Three.js

Now that we've introduced Three.js and TypeScript, let's go a step further and see how we can integrate this with React as well.

Using Three.js and React with TypeScript

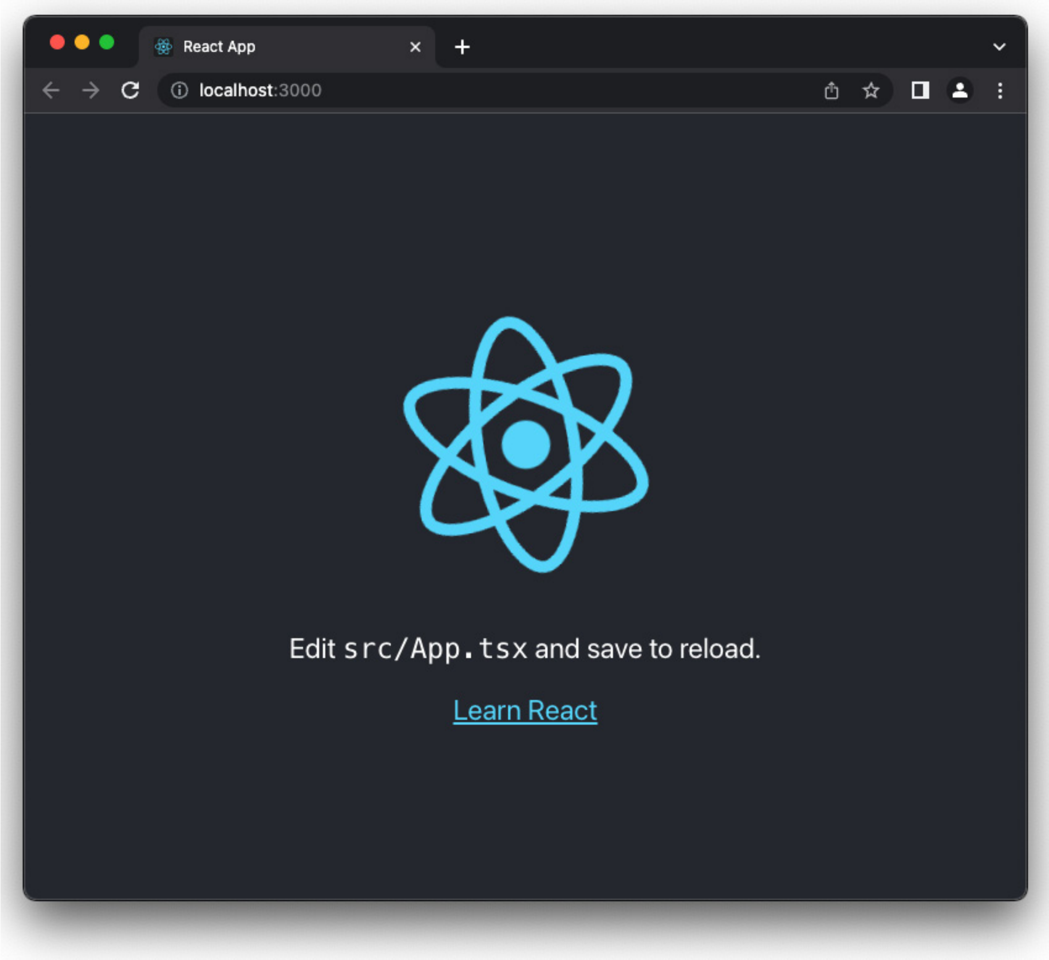
There are different ways of creating a React application from scratch (Vite, for instance, also supports this), but the most common way is to use the `yarn create react-app lts-tf --template TypeScript` command from the command line. Just like with Vite, this will create a new project. For this example, we've created this project in the `lts-tf` directory. Once created, we have to add the Three.js libraries just like we did for Vite:

```
$ yarn create react-app lts-tf --template TypeScript
...
$ cd lts-tf
$ yarn add three
$ yarn add -D @types/three
$ yarn install
```

This should set up a simple react TypeScript application, add the correct Three.js libraries, and install all the other required modules. The next step is to quickly check if all this works. Run the `yarn start` command:

```
$ yarn start
Compiled successfully!
You can now view lts-tf in the browser.
  Local:            http://localhost:3000
  On Your Network:  http://192.168.68.112:3000
Note that the development build is not optimized.
To create a production build, use yarn build.
webpack compiled successfully
Files successfully emitted, waiting for typecheck results...
Issues checking in progress...
No issues found.
```

Open your browser to `http://localhost:3000` and you'll see a simple React startup screen:



Edit `src/App.tsx` and save to reload.

Learn React

Figure 14.3 – Simple TypeScript project with Three.js

On this screen, we can see that we need to edit the `app.tsx` file, so we'll update this similar to the plain TypeScript example we saw in the *Using Three.js with TypeScript* section, but this time as a React component:

```
import './App.css'
import { ThreeCanvas } from './ThreeCanvas'
function App() {
  return (
    <div className="App">
```

```
      <ThreeCanvas></ThreeCanvas>
    </div>
  )
}
export default App
```

As you can see, here, we defined a custom component named `ThreeCanvas`, which is now loaded as soon as the application starts. The Three.js initialization code is provided by the `ThreeCanvas` element, which you can find in the `ThreeCanvas.tsx` file. This file, for the most part, is similar to the `initThreeJsScene` function we described in the *Using Three.js with TypeScript* section, but we'll include the whole file here for completeness:

```
import { useCallback, useState } from 'react'
import * as THREE from 'three'
const initThreeJsScene = (node: HTMLDivElement) => {
  const scene = new THREE.Scene()
  const camera = new THREE.PerspectiveCamera(75, 500 / 500,
    0.1, 1000)
  const renderer = new THREE.WebGLRenderer()
  renderer.setClearColor(0xffffff)
  renderer.setSize(500, 500)
  node.appendChild(renderer.domElement)
  camera.position.z = 5
  const geometry = new THREE.BoxGeometry()
  const material = new THREE.MeshNormalMaterial()
  const cube = new THREE.Mesh(geometry, material)
  scene.add(cube)
  const animate = () => {
    requestAnimationFrame(animate)
    cube.rotation.x += 0.01
    cube.rotation.y += 0.01
    renderer.render(scene, camera)
  }
  animate()
}
export const ThreeCanvas = () => {
  const [initialized, setInitialized] = useState(false)
  const threeDivRef = useCallback(
    (node: HTMLDivElement | null) => {
      if (node !== null && !initialized) {
        initThreeJsScene(node)
        setInitialized(true)
      }
    },
    [initialized]
  )
  return (
    <div
      style={{
        display: 'flex',
        alignItems: 'center',
        justifyContent: 'center',
        height: '100vh'
      }}
      ref={threeDivRef}
    ></div>
  )
}
```

In `initThreeJsScene`, you can find the standard code to initialize a simple Three.js scene using TypeScript. To connect this Three.js scene to React, we can use the code from the `ThreeCanvas` functional React component. What we want to do here is initialize the Three.js scene at the moment the `div` element gets attached to its parent node. To do this, we can use the `useCallback` function. This function will be called once when this node gets attached to its parent, and won't rerun even if one of the parent properties changes. In our case, we will also add another `isInitialized` state to make sure that even if we have the development server reload parts of the application, we only initialize our Three.js scene once.

useRef or useCallback

You might be tempted to use `useRef` here. There is a good explanation at `https://reactjs.org/docs/hooks-faq.html#how-can-i-measure-a-dom-node` regarding why, in this case, you should use `useCallback` instead of `useRef` to avoid unnecessary rerenderings.

With the preceding setup in place, we can now see the result:

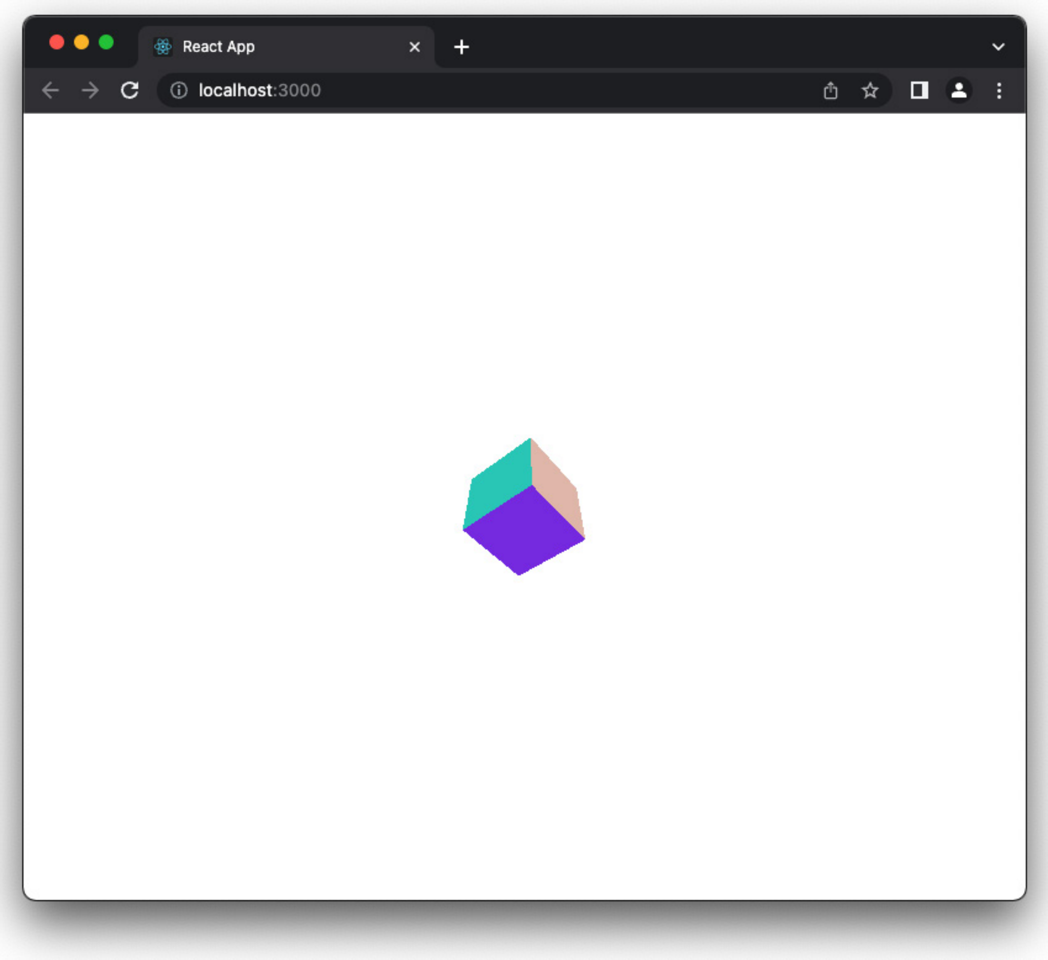

React App
localhost:3000

Figure 14.4 – Three.js with TypeScript and React

In the previous example, we created a simple integration between React and Three.js. While this works, it feels a bit strange to programmatically describe a Three.js scene, since normally, in React, applications are declared declaratively using components. We can wrap the existing Three.js components as we did with the `ThreeCanvas` component, but this will quickly get complex. Luckily, though, all the hard work for this has already been done by the Three.js fibers project: `https://docs.pmnd.rs/react-three-fiber/getting-started/introduction`. In the next section, we'll look at how easily Three.js and React can be integrated with the help of this project.

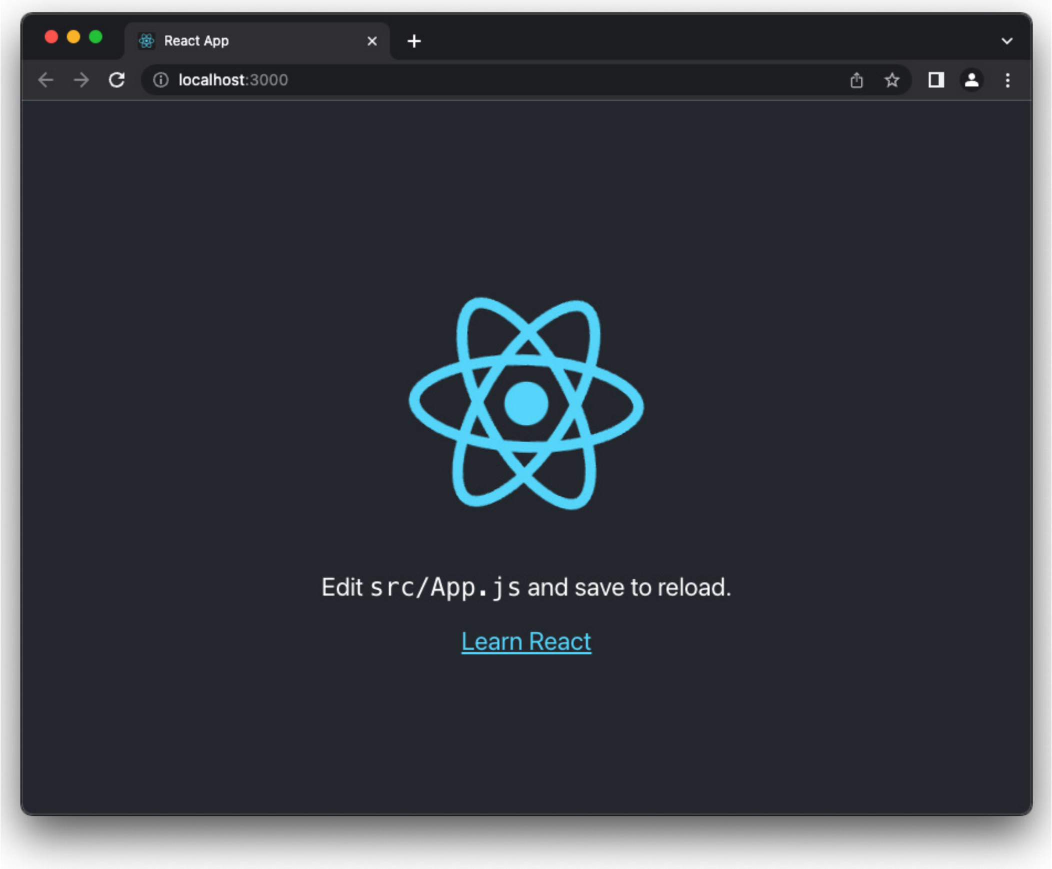
Using Three.js and React with React Three Fiber

In the previous examples, we set up the integration between React and Three.js ourselves. While it works, that approach doesn't tightly integrate with how React works. For a good integration between these frameworks, we can use React Three Fiber. We'll start again by setting up a project.

For this, run the following commands:

```
$ yarn create react-app lts-r3f
$ cd lts-3rf
$ yarn install
$ yarn add three
$ yarn add @react-three/fiber
```

This will install all the dependencies we need and set up a new React project. To start this project in the `lts-r3f` directory, run `yarn start`, which will start a server. Open the URL you see on the screen (`http://localhost:3000`); you'll see the following screen, which we've seen before and shows an empty React project:



Edit `src/App.js` and save to reload.

Learn React

Figure 14.5 – Starting up a simple JavaScript React application

As the screen starts to extend this example, we need to edit the `app.jsx` file. We'll create a new component that will contain our Three.js scene:

```
import './App.css'
import { Canvas } from '@react-three/fiber'
import { Scene } from './Scene'
function App() {
  return (
    <Canvas>
      <Scene />
    </Canvas>
  )
}
export default App
```

Here, we can already see the first of the Three Fiber components – the `Canvas` element. The `Canvas` element creates a `Canvas` div and is the parent container for all the other Three.js components provided by this library. Since we added `Scene` as a child to this `Canvas` component, we can define our complete Three.js scene in our custom component. Next, we'll create this `Scene` component:

```
import React from 'react'
export const Scene = () => {
  return (
    <>
      <ambientLight intensity={0.1} />
      <directionalLight color="white" intensity={0.2}
       position={[0, 0, 5]} />
      <mesh
        rotation={[0.3, 0.6, 0.3]}>
        <boxGeometry args={[2, 5, 1]} />
        <meshStandardMaterial color={color}
          opacity={opacity} transparent={true} />
      </mesh>
    </>
  )
}
```

What we've got here is a very simple Three.js scene, which looks similar to the ones we saw earlier in this book. This scene contains the following objects:

- `<ambientLight>`: An instance of a `Three.AmbientLight` object.
- `<directionalLight>`: An instance of a `Three.DirectionalLight` object.
- `<mesh>`: This represents a `Three.Mesh`. As we know, a `Three.Mesh` contains a geometry and a material, which are defined as children of this element. In this example, we've set the rotation on this mesh as well.
- `<boxGeometry>`: This is similar to `Three.BoxGeometry`, where we pass in the constructor arguments through the `args` property.

- `<meshStandardMaterial>`: This creates an instance of a `THREE.MeshStandardMaterial`, and configures some properties on this material.

Now, when you open your browser to `localhost:3000`, you'll see a Three.js scene:

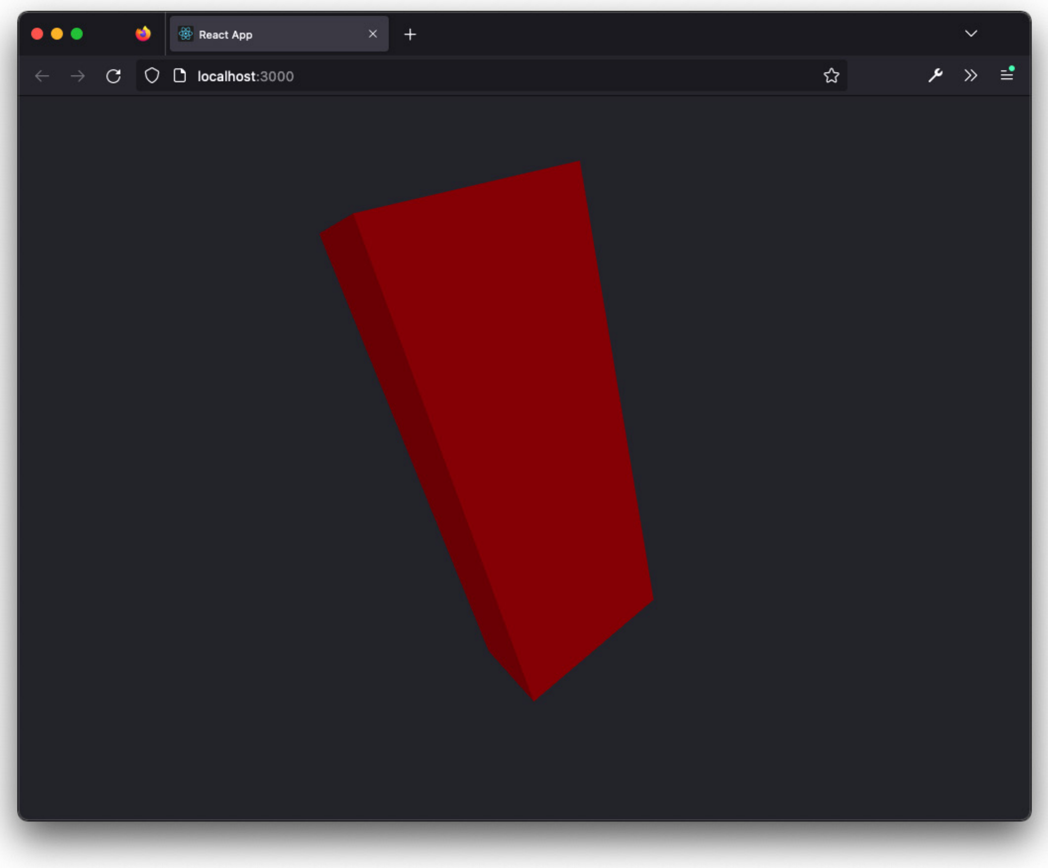
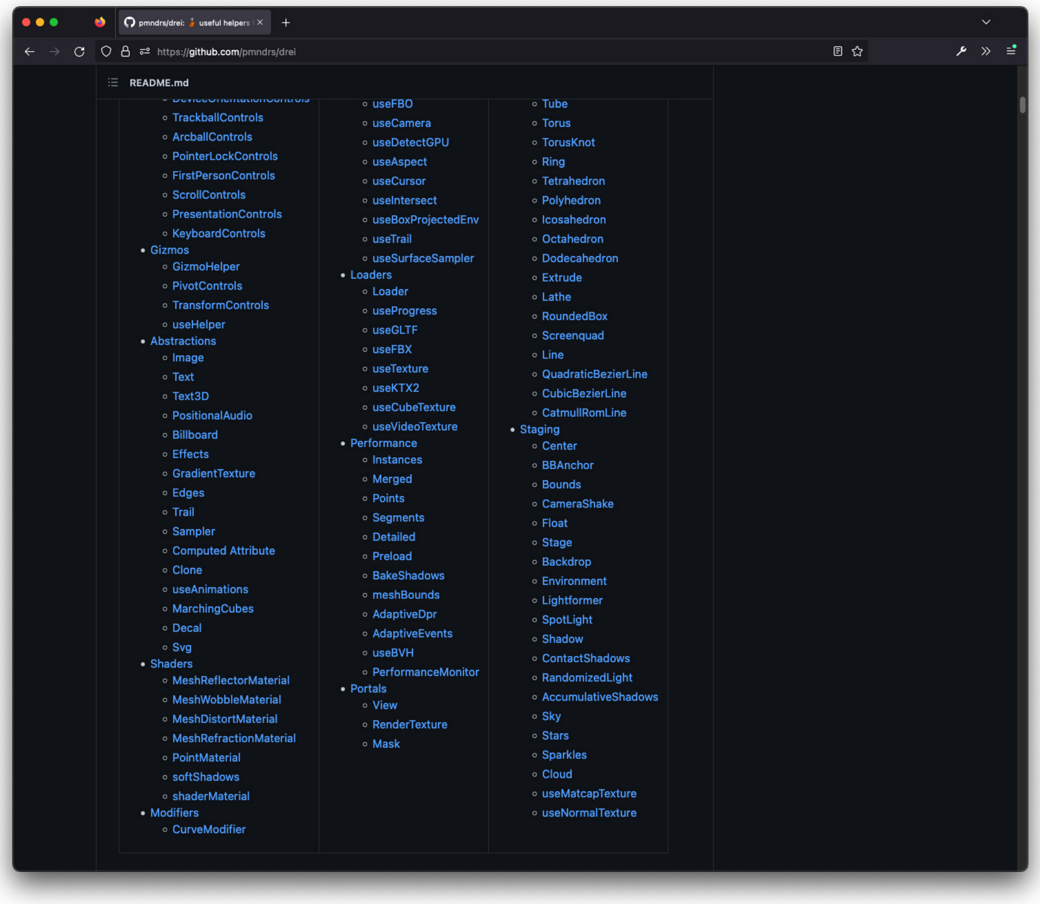


Figure 14.6 – Three.js scene rendered using React Three Fiber

In this example, we've only shown a couple of small elements provided by React Three Fiber. All objects provided by Three.js can be configured in the way we just showed. Just add them as elements, configure them, and they will be shown on the screen. Besides easily showing these elements, all these elements behave like normal React components. So, whenever the properties of a parent element change, all the elements are rerendered (or updated) as well.

Besides the elements provided by React Three Fiber, there are a whole set of additional components provided by `@react-three/drei`. You can find those components and their descriptions at `https://github.com/pmndrs/drei`:



  - TrackballControls
  - ArcballControls
  - PointerLockControls
  - FirstPersonControls
  - ScrollControls
  - PresentationControls
  - KeyboardControls
- Gizmos
  - GizmoHelper
  - PivotControls
  - TransformControls
  - useHelper
- Abstractions
  - Image
  - Text
  - Text3D
  - PositionalAudio
  - Billboard
  - Effects
  - GradientTexture
  - Edges
  - Trail
  - Sampler
  - Computed Attribute
  - Clone
  - useAnimations
  - MarchingCubes
  - Decal
  - Svg
- Shaders
  - MeshReflectorMaterial
  - MeshWobbleMaterial
  - MeshDistortMaterial
  - MeshRefractionMaterial
  - PointMaterial
  - softShadows
  - shaderMaterial
- Modifiers
  - CurveModifier

  - useFBO
  - useCamera
  - useDetectGPU
  - useAspect
  - useCursor
  - useIntersect
  - useBoxProjectedEnv
  - useTrail
  - useSurfaceSampler
- Loaders
  - Loader
  - useProgress
  - useGLTF
  - useFBX
  - useTexture
  - useKTX2
  - useCubeTexture
  - useVideoTexture
- Performance
  - Instances
  - Merged
  - Points
  - Segments
  - Detailed
  - Preload
  - BakeShadows
  - meshBounds
  - AdaptiveDpr
  - AdaptiveEvents
  - useBVH
  - PerformanceMonitor
- Portals
  - View
  - RenderTexture
  - Mask

  - Tube
  - Torus
  - TorusKnot
  - Ring
  - Tetrahedron
  - Polyhedron
  - Icosahedron
  - Octahedron
  - Dodecahedron
  - Extrude
  - Lathe
  - RoundedBox
  - Screenquad
  - Line
  - QuadraticBezierLine
  - CubicBezierLine
  - CatmullRomLine
- Staging
  - Center
  - BBAnchor
  - Bounds
  - CameraShake
  - Float
  - Stage
  - Backdrop
  - Environment
  - Lightformer
  - SpotLight
  - Shadow
  - ContactShadows
  - RandomizedLight
  - AccumulativeShadows
  - Sky
  - Stars
  - Sparkles
  - Cloud
  - useMatcapTexture
  - useNormalTexture

Figure 14.7 – Additional components from @react-three/drei

For the next example, we're going to use a couple of the components provided by this library, so we need to add this to our project:

```
$ yarn add @react-three/drei
```

Now, we'll extend our example to this:

```
import React, { useState } from 'react'
import './App.css'
import { OrbitControls, Sky } from '@react-three/drei'
import { useFrame } from '@react-three/fiber'
```

```
export const Scene = () => {
  // run on each render of react
  // const size = useThree((state) => state.size)
  const mesh = React.useRef()
  const [color, setColor] = useState('red')
  const [opacity, setOpacity] = useState(1)
  const [isRotating, setIsRotating] = useState(false)
  // run on each rerender of
  useFrame(({ clock }, delta, xrFrame) => {
    if (isRotating) mesh.current.rotation.x += 0.01
  })
  return (
    <>
      <Sky distance={450000} sunPosition={[0, 1, 0]}
        inclination={0} azimuth={0.25} />
      <ambientLight intensity={0.1} />
      <directionalLight color="white" intensity={0.2}
        position={[0, 0, 5]} />
      <OrbitControls></OrbitControls>
      <mesh
        ref={mesh}
        rotation={[0.3, 0.6, 0.3]}
        onClick={() => setColor('yellow')}
        onPointerEnter={() => {
          setOpacity(0.5)
          setIsRotating(true)
        }}
        onPointerLeave={() => {
          setOpacity(1)
          setIsRotating(false)
        }}
      >
        <boxGeometry args={[2, 5, 1]} />
        <meshStandardMaterial color={color}
          opacity={opacity} transparent={true} />
      </mesh>
    </>
  )
}
```

Let's explore the code a bit before looking at the result in the browser. First, we'll look at the new elements we added to the component:

- `<OrbitControls>`: This is provided by the `drei` library. This adds an `THREE.OrbitControls` element to the scene. This is the same as `OrbitControls`, which we used in the earlier chapters. As you can see here, just adding the element is enough; no additional configuration is needed.
- `<Sky>`: This element provides a nice sky background to the scene.

We've also added several standard React Hooks:

```
  const mesh = React.useRef()
  const [color, setColor] = useState('red')
  const [opacity, setOpacity] = useState(1)
  const [isRotating, setIsRotating] = useState(false)
```

Here, we define a `Ref`, which we connect to the mesh (`<mesh ref={mesh}) ..>`). We use this so that we can access the Three.js component later on in the render loop. We also use `useState` three times to keep track of the `color` and `opacity` state values of the material, as well as to see whether the `mesh` property is rotating or not. The first of these two Hooks is used in the events we defined on the mesh:

```
      <mesh
        onClick={() => setColor('yellow')}
        onPointerEnter={() => {
          setOpacity(0.5)
          setIsRotating(true)
        }}
        onPointerLeave={() => {
          setOpacity(1)
          setIsRotating(false)
        }}>
```

With these event handlers, we can very easily integrate the mouse with the mesh. There's no need for `RayCaster` objects – just add the event listener and you're done. In this case, when the mouse pointer enters our mesh, we change the `opacity` state value and the `isRotation` flag. When the mouse leaves our mesh, we set the `opacity` state value back and set the `isRotation` flag to `false` again. Finally, when we click on the mesh, we change the color to `yellow`.

The `color` and `opacity` state values can be directly used in `meshStandardMaterial` like this:

```
<meshStandardMaterial color={color} opacity={opacity}
  transparent={true} />
```

Now, the opacity and color will automatically update whenever we fire the relevant events. For the rotation, we want to use the Three.js render loop. For this, React Three Fiber provides an additional hook:

```
  useFrame(({ clock }, delta, xrFrame) => {
    if (isRotating) mesh.current.rotation.x += 0.01
  })
```

`useFrame` is called whenever we have a render loop in Three.js. In this case, we check the `isRotating` state, and if we should be rotating, we use the previously defined `useRef` reference to get access to the underlying Three.js component and simply increase its rotation. This is all very easy and convenient. The result in the browser looks like this:

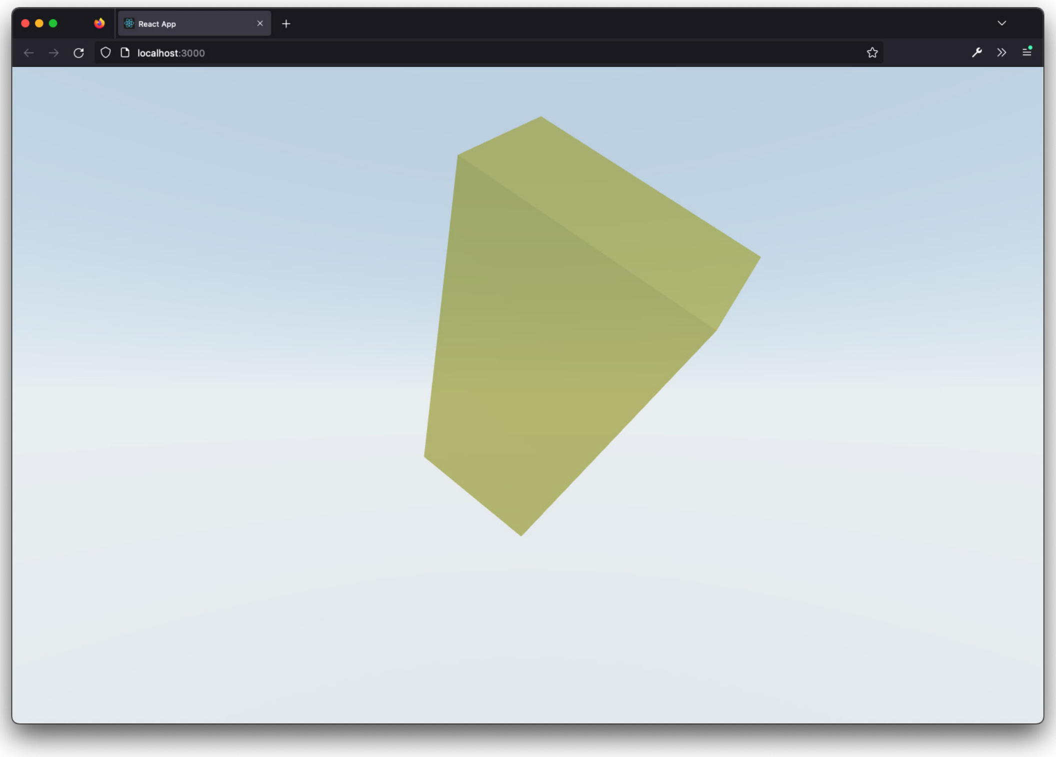


Figure 14.8 – A scene using React and React Three Fiber effects

React Three Fiber and the `drei` library provide pretty much all the functionality you also have in the normal Three.js library (and some features that aren't available as well). If you're working with React and need Three.js integrated, this is a great way to use Three.js. Even when you are not necessarily building a React application, the declarative way of defining your scene, the components, and the interactions offered by React Three Fiber is very intuitive. React Three Fiber provides a great alternative for any Three.js visualization you want to create.

In the next two sections, we'll look at how you can extend your 3D scenes with AR and VR capabilities. We'll start by looking at how to enable VR in your scenes.

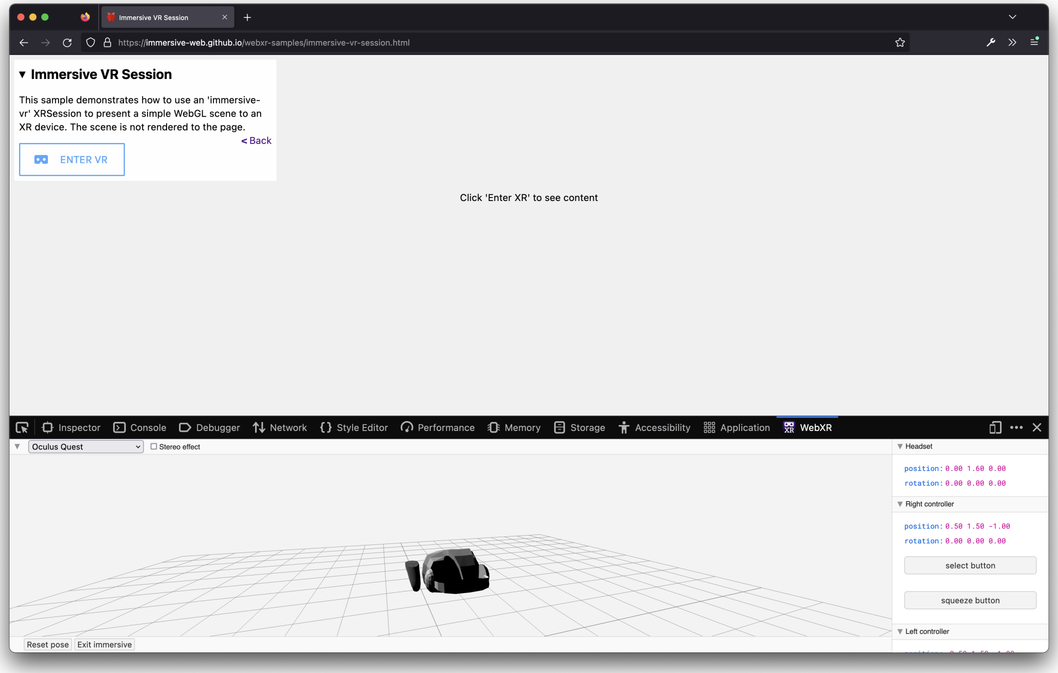
Three.js and VR

Before we look at the required code changes, we'll add an extension to the browser with which we can simulate a VR headset and VR controls. That way, you can test your scenes without the need for a physical headset and physical controllers. For this, we'll install the WebXR API simulator This plugin is available for both Firefox and Chrome:

- **Firefox plugin**: Download and install it from here: `https://addons.mozilla.org/en-US/firefox/addon/webxr-api-emulator/`
- **Chrome plugin**: Download and install it from here: `https://chrome.google.com/webstore/detail/webxr-api-emulator/mjddjgeghkdijejnciaefnkjmkafnnje`

Follow the instructions for your specific browser. After you've installed it, we can test it with this example: `https://immersive-web.github.io/webxr-samples/immersive-vr-session.html`.

Open this example, open your developer console, and click on the **WebXR** tab. Now, you'll see something like this:



https://immersive-web.github.io/webxr-samples/immersive-vr-session.html

▼ Immersive VR Session

This sample demonstrates how to use an 'immersive-vr' XRSession to present a simple WebGL scene to an XR device. The scene is not rendered to the page.

Click 'Enter XR' to see content

Oculus Quest

☐ Stereo effect

▼ Headset

position: 0.00 1.60 0.00
rotation: 0.00 0.00 0.00

▼ Right controller

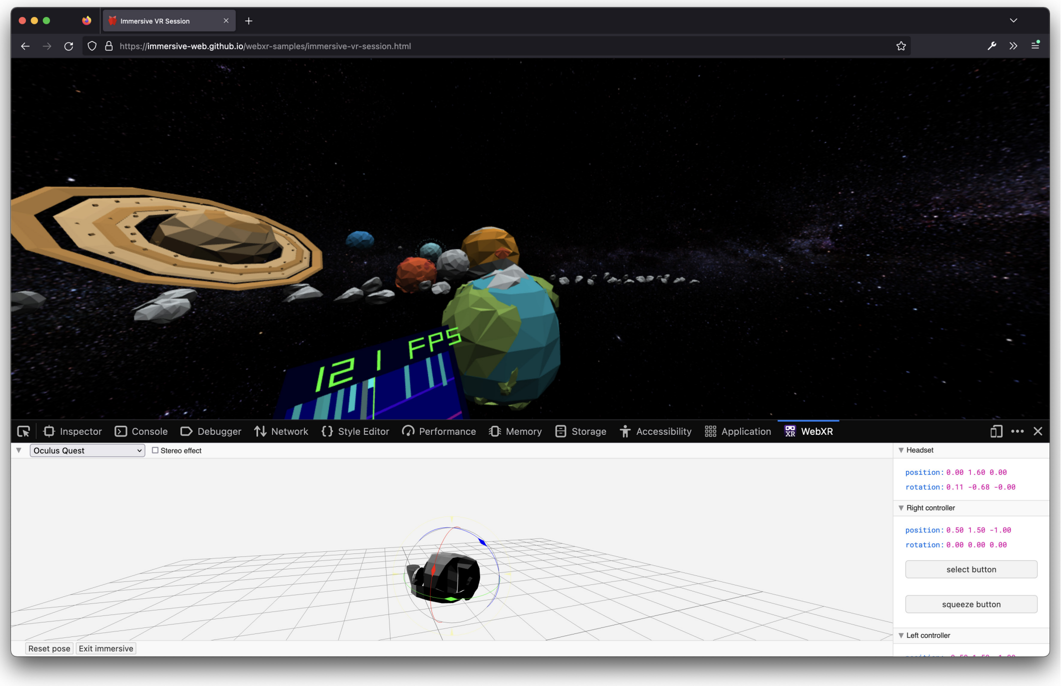
position: 0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

▼ Left controller

Immersive VR Session
< Back
ENTER VR
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
WebXR
select button
squeeze button
Reset pose
Exit immersive

Figure 14.9 – Firefox browser with the WebXR API extension

In the extension, you'll see a virtual headset and some virtual VR controls. By clicking on the headset, you can simulate the movement of a real VR headset; the same applies to the controls. If you click the **Enter VR** button, you can now simply test your VR scenes, without an actual VR headset:

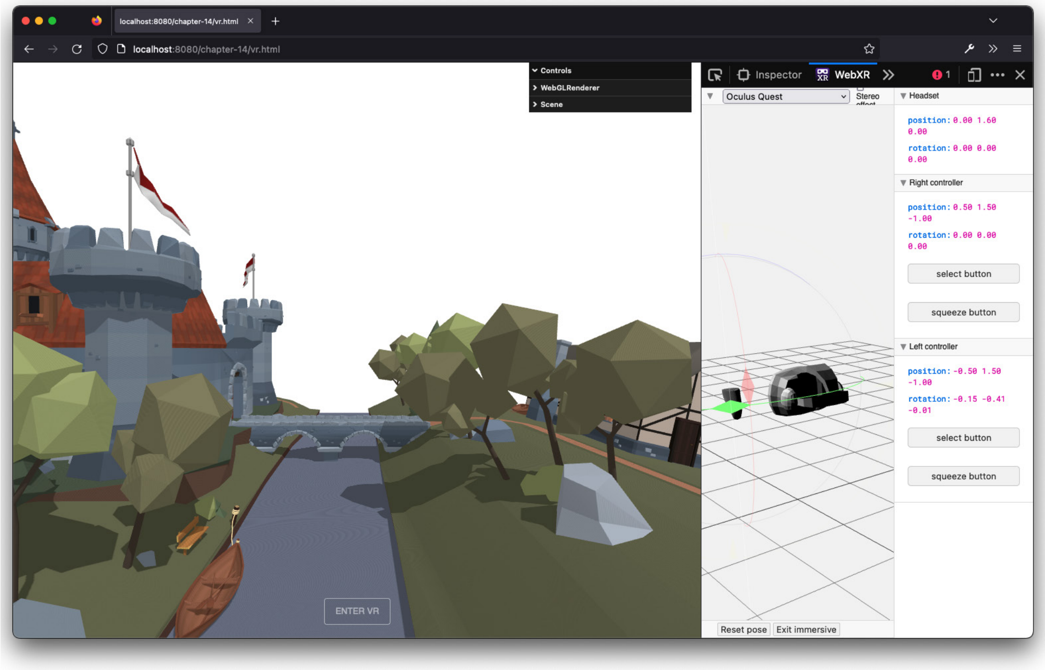

Immersive VR Session
https://immersive-web.github.io/webxr-samples/immersive-vr-session.html
121 FPS
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
WebXR
Oculus Quest
☐ Stereo effect
Headset
Right controller
select button
squeeze button
Left controller
Reset pose
Exit immersive

position: 0.00 1.60 0.00
rotation: 0.11 -0.68 -0.00

position: 0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

Figure 14.10 – Simulating a VR headset

Now that we've got a (virtual) headset to play around with, let's convert one of our previous scenes into a VR scene, where we can track head movement and add functionality to some dummy VR controls. For this, we've created a copy of the "First Person Controls" example from *Chapter 8*, *Creating and Loading Advanced Meshes and Geometries*. You can open this example by opening up the `vr.html` example from the `chapter-14` sources:


localhost:8080/chapter-14/vr.html
localhost:8080/chapter-14/vr.html
Controls
WebGLRenderer
Scene
ENTER VR
Inspector
WebXR
1
Oculus Quest
Stereo effect
Headset
Right controller
select button
squeeze button
Left controller
select button
squeeze button
Reset pose
Exit immersive

Headset
position: 0.00 1.60 0.00
rotation: 0.00 0.00 0.00

Right controller
position: 0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

Left controller
position: -0.50 1.50 -1.00
rotation: -0.15 -0.41 -0.01

Figure 14.11 – Empty VR scene based on example from Chapter 9

To get your scene VR ready, we need to take a couple of steps. The first thing we need to do is tell Three.js that we'll be enabling the Web-XR functionality. This can be done like this:

```
renderer.xr.enabled = true
```

The next step is to add a simple button that we can click to enter VR mode. Three.js provides a component out of the box for this, which we can use like this:

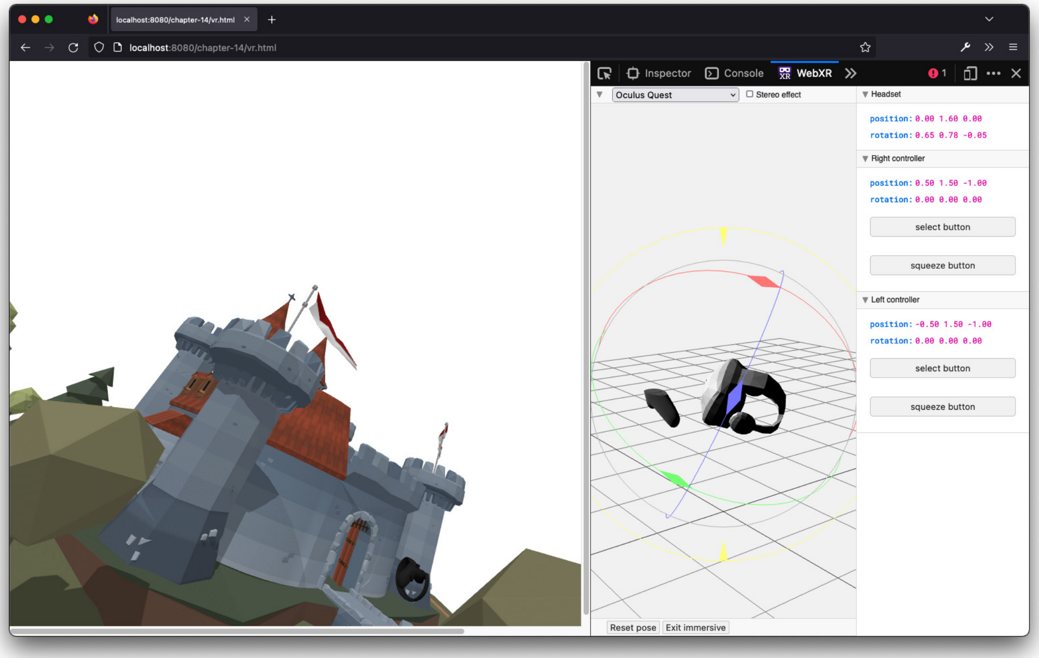
```
import { VRButton } from 'three/examples/jsm/webxr/VRButton'
document.body.appendChild(VRButton.createButton(renderer))
```

This will create the button you can see at the bottom of the screen in *Figure 14.11*.

Finally, we need to update our render loop. As you may recall from *Chapter 1*, *Creating Your First 3D Scene With Three.js*, we use `requestAnimationFrame` to control our render loop. When working with VR, we need to change this slightly, like so:

```
animate()
function animate() {
  renderer.setAnimationLoop(animate)
  renderer.render(scene, camera)
  if (onRender) onRender(clock, controls, camera, scene)
}
```

Here, we used `renderer.setAnimationLoop` instead of `requestAnimationFrame`. At this point, our scene has been converted into VR, and once we click the button, we enter VR mode and can look around our scene:


localhost:8080/chapter-14/vr.html

Oculus Quest
☐ Stereo effect

Headset
position: 0.00 1.60 0.00
rotation: 0.65 0.78 -0.05

Right controller
position: 0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

Left controller
position: -0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

Figure 14.12 – Entering VR mode and rotating the camera using the browser extension

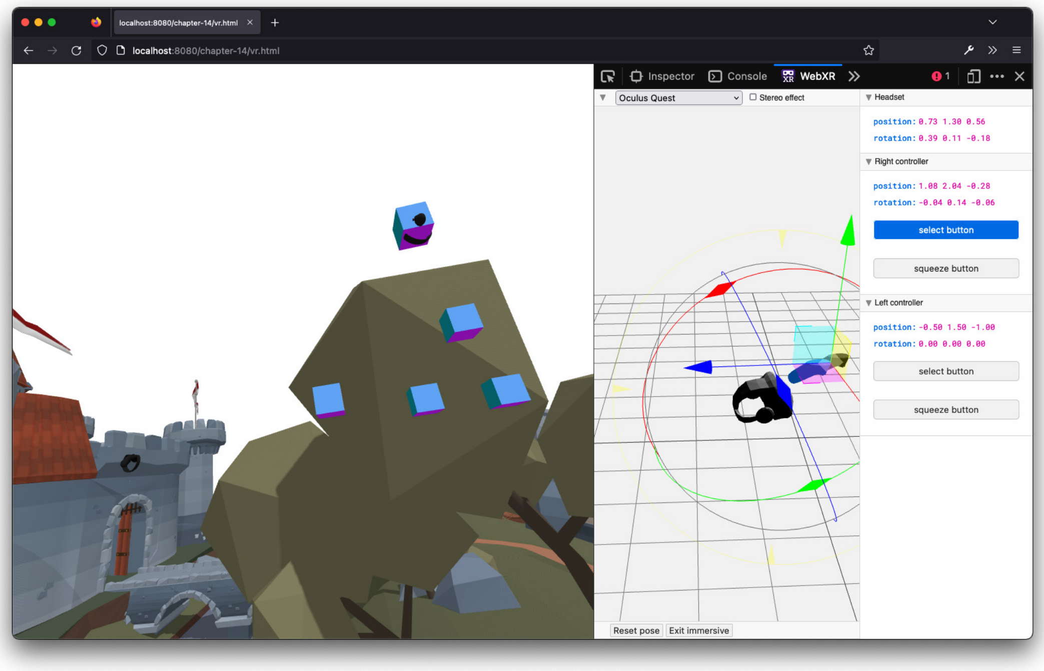
The previous screenshot is shown when you enter VR. You can now easily move the camera around by clicking on the VR device in the Web-XR extension and move it around. These steps are pretty much all you need to do to convert any Three.js scene into a VR scene.

If you look closely at *Figure 14.12*, you might notice that we also show some handheld VR devices. We haven't shown how you can add these yet. For this, Three.js also comes with some nice helper components:

```
import { XRControllerModelFactory } from
  'three/examples/jsm/webxr/XRControllerModelFactory'
const controllerModelFactory = new
  XRControllerModelFactory()
const controllerGrip1 = renderer.xr.getControllerGrip(0)
controllerGrip1.add(controllerModelFactory.
createControllerModel(controllerGrip1))
scene.add(controllerGrip1)
const controllerGrip2 = renderer.xr.getControllerGrip(1)
controllerGrip2.add(controllerModelFactory.
createControllerModel(controllerGrip2))
scene.add(controllerGrip2)
```

With the preceding code, we ask Three.js to get information about the attached controllers, create a model, and add them to the scene. If you use the WebXR API emulator, you can move the controls around and they'll move around on the scene as well.

Three.js provides lots of examples of how you can use these controls to drag objects around, select objects in the scene, and otherwise add interactivity with the controls. For this simple example, we've added the option to add a cube at the position of the first control (the one on the right) whenever you click the `select` button:


localhost:8080/chapter-14/vr.html
localhost:8080/chapter-14/vr.html
Inspector
Console
WebXR
1
Oculus Quest
☐ Stereo effect
Reset pose
Exit immersive
select button
squeeze button
select button
squeeze button

Headset

position: 0.73 1.30 0.56
rotation: 0.39 0.11 -0.18

Right controller

position: 1.08 2.04 -0.28
rotation: -0.04 0.14 -0.06

Left controller

position: -0.50 1.50 -1.00
rotation: 0.00 0.00 0.00

Figure 14.13 – Adding cubes to the scene in VR

We can do this by simply adding event listeners to the controller, like this:

```
const controller = renderer.xr.getController(0)
controller.addEventListener('selectstart', () => {
  console.log('start', controller)
  const mesh = new THREE.Mesh(new THREE.BoxGeometry(0.1,
    0.1, 0.1), new THREE.MeshNormalMaterial())
  mesh.position.copy(controller.position)
  scene.add(mesh)
})
controller.addEventListener('selectend', () => {
  console.log('end', controller)
})
```

In this simple piece of code, you can see that we've added two event listeners to the controller. When the `selectstart` event is triggered, we add a new cube to the location of this controller. And when the `selectend` event is triggered, we just log some information to the console. Several other events can be accessed through JavaScript. For more information about the APIs that are available when you are in a VR session, you can look at the following documentation: `https://developer.mozilla.org/en-US/docs/Web/API/XRSession`

For the last section, we'll have a quick look at how you can combine Three.js with AR.

Three.js and AR

While VR using Three.js is well supported on a large range of devices and browsers, this isn't the case for Web-AR. On Android, the support is pretty good, but on iOS devices, it doesn't work that well. Apple is currently working on adding this to Safari, so once that's in, native AR should also work on iOS. A good way to check which browsers support this functionality is to check `https://caniuse.com/webxr`, which provides an up-to-date overview of all major browser support.

So, to test the native AR example, you either need to view it on an Android device, or use the same simulator we used in the *Three.js and VR* section.

Let's create a standard scene you can use as a starting point for your AR experiments. The first thing we need to do is tell Three.js we want to use XR:

```
const renderer = new THREE.WebGLRenderer({ antialias: true,
  alpha: true })
renderer.xr.enabled = true
```

Note that we need to set the `alpha` property to `true`; otherwise, we won't see any passthrough from the camera. Next, just like we did for VR, we need to enable AR/VR on the renderer, by calling `rendered.xr.enabled`.

To enter AR mode, Three.js also provides a button we can use:

```
Import { ARButton } from 'three/examples/jsm/
  webxr/ARButton'
document.body.appendChild(ARButton.createButton(renderer))
```

Finally, we just need to change `requestAnimationFrame` to `setAnimationLoop`:

```
animate()
function animate() {
  renderer.setAnimationLoop(animate)
  renderer.render(scene, camera)
}
```

And that's all there is to it. If you open the `ar.html` example and view this example through the WebXR plugin (where you need to select the `Samsung Galaxy S8+ (AR)` device), you'll see something like this:

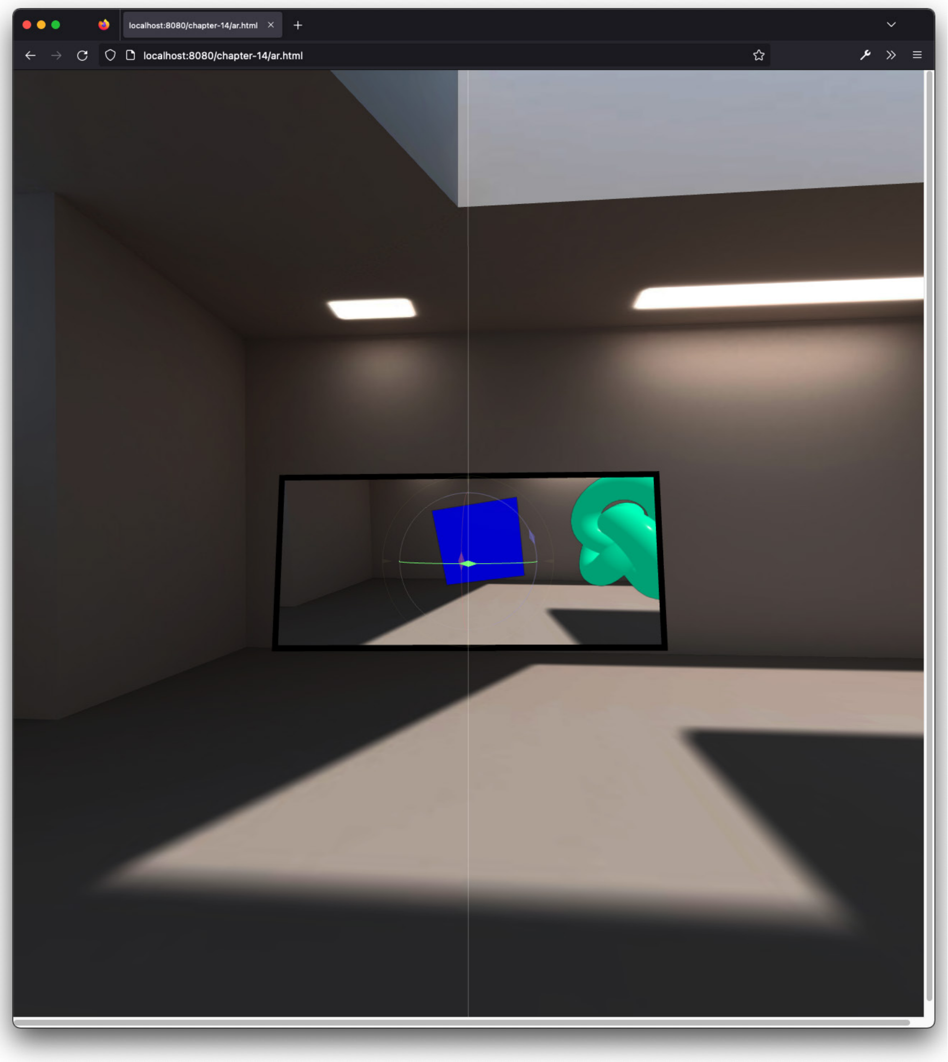

localhost:8080/chapter-14/ar.html
localhost:8080/chapter-14/ar.html

Figure 14.14 – Viewing an AR scene in Three.js using a device's native AR functionality

In this screenshot, you can see a simulated AR environment, where we can see the two objects that we've rendered. If you move the simulated phone around, you'll notice that the rendered objects are fixed in their location relative to the position of the camera of the phone.

The example here is very simple but it shows the basics of how to set up a minimal AR scene. Web-XR offers a lot of other functionality related to AR, such as detecting planes and hit testing. However, covering that falls a bit outside of the scope of this book. For more information on Web-XR and the native AR functionality exposed by this API, you can look at the specifications here: `https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API/Fundamentals`.

Summary

In this chapter, we looked at a couple of technologies related to Three.js. We showed you different ways of integrating Three.js with TypeScript and React, and we also showed you how to create some basic AR and VR scenes.

By using the Three.js TypeScript bindings, you can easily access all the Three.js functionality from your TypeScript project. And integrating Three.js with React is made easy through the React Three Fiber library.

Using VR and AR in Three.js is also very straightforward. By just adding a couple of properties to the main renderer, you can quickly convert any scene into a VR or AR scene. Remember to use the browser plugin to easily test your scenes without needing actual VR and AR devices.

With that, we've come to the end of this book. I hope you've enjoyed reading it and playing around with the examples. Happy experimenting!

Index

Symbols

2D geometries 148
  THREE.CircleGeometry 151-153
  THREE.PlaneGeometry 149-151
  THREE.RingGeometry 153, 154
  THREE.ShapeGeometry 155-159
3D geometries 159
  THREE.BoxGeometry 160, 161
  THREE.ConeGeometry 166, 167
  THREE.CylinderGeometry 164, 165
  THREE.DodecahedronGeometry 176, 177
  THREE.IcosahedronGeometry 173
  THREE.OctahedronGeometry 175, 176
  THREE.PolyhedronGeometry 171-173
  THREE.SphereGeometry 161-163
  THREE.TetrahedronGeometry 174
  THREE.TorusGeometry 167-169
  THREE.TorusKnotGeometry 169, 170
3DM 241
  URL 241
3D object
  animation loop, adding 22
  lights, adding 19
  lil-gui library, experimenting 28, 29
  lil-gui library, using to control properties 28, 29
  meshes, adding 20-22
  rendering 15, 16
  setting up 16-18
  viewing 15, 16
3DS 242
  URL 242
3D shapes
  extruding, from SVG elements 192-195
3D text mesh
  creating 201
  custom fonts, adding 204, 205
  text, creating with Troika library 205-207
  text, rendering 202-204
3MF 241
  URL 241

A

advanced geometries
  BoxLineGeometry 184, 185
  TeapotGeometry 186, 187
  THREE.ConvexGeometry 180-182
  THREE.LatheGeometry 182, 183
  THREE.RoundedBoxGeometry 185

advanced materials 123
  shaders, customizing with CustomShaderMaterial 139, 140
  shaders, using with THREE.ShaderMaterial 133-139
  THREE.MeshLambertMaterial 124-126
  THREE.MeshPhongMaterial 127
  THREE.MeshPhysicalMaterial 131
  THREE.MeshStandardMaterial 129, 130
  THREE.MeshToonMaterial 128, 129
  THREE.ShadowMaterial 132
alpha map
  using, to create transparent models 339-341
ambient occlusion map 390-392
  baking, in Blender 464, 478-480
  subtle shadows, adding with 331-334
AMF 241
  URL 241
anaglyph effect 57
animation
  exporting, from Blender 454-463
  importing, into Three.js 454-463
  loading, from COLLADA model 314
  loading, from Quake model 313
  with bones and skinning 304-308
  with mixer and morph targets 297-300
  with morph targets 296
animation loop
  meshes, animating 24, 25
  orbit controls, enabling 26, 27
  requestAnimationFrame 23, 24
ArcballControls 283, 284
  methods 286
  properties 285
augmented reality (AR) 3
AWS Cloud9
  reference link 7

B

basic animations 267, 268
basic lights
  THREE.AmbientLight 69-72
  THREE.Color object 87-89
  THREE.DirectionalLight 84-87
  THREE.PointLight 80-84
  THREE.SpotLight 72-80
billboarding 226
Biovision (BVH) format 308
Blender
  ambient occlusion maps, baking 464, 478-480
  animation, exporting from 454-463
  lightmaps, baking 464
  scene, exporting from 475-478
  scene, importing into 444-450, 475-478
  scene, setting up 464-466
  static scene, exporting from 450-454
  UV modeling 480-486
BloomPass
  properties 377
blurring shaders 397-399
  FocusShader 399
  HorizontalBlurShader 397
  HorizontalTiltShiftShader 398
  VerticalBlurShader 397
  VerticalTiltShiftShader 398
BokehPass 387
bones and skinning
  animating with 304-308
BoxLineGeometry 184, 185
BrightnessContrastShader effect 394
Building Information Modeling (BIM) tools 241

bump map
  canvas, using as 359-361
  using, to add details to mesh 324-326
BVHLoader
  skeleton, visualizing 316, 317

C

camera 33
  ArcballControls 283-286
  FirstPersonControls 283, 290-292
  FlyControls 283, 289, 290
  OrbitControls 283, 292-294
  PointerLockControls 283
  TrackballControls 283-288
  working with 283
cameras, for scene
  position, pointing 63, 64
  visuals, debugging 64, 65
canvas
  rendering to 357
  using, as bump map 359-361
  using, as color map 357-359
Chrome plugin
  download link 508
CNC machines 241
COLLADA 241, 308
  animation, loading from 314
color map
  canvas, using as 357-359
Constructive Solid Geometry (CSG) 240
  geometries, creating through 240
custom postprocessing shaders
  creating 399
  custom bit shader, creating 404, 405
  custom grayscale shader, creating 400-404
CustomShaderMaterial
  used, for customizing existing shaders 139, 140

D

displacement map
  using, to alter vertices position 329-331
dominos
  simulating, in Rapier 415-421
DotScreenPass
  properties 379
Draco 241
  URL 241

E

easing 278
EffectComposer
  RenderPass 375
  ShaderPass with CopyShader 375
  ShaderPass with GammaCorrectionShader 375
EffectComposer flows, with masks 382-387
  ambient occlusion, adding 390-392
  bokeh effect, adding with BokehPass 387-390
effect passes
  reference link 369
emissive map
  using, for models that glow 342-344
environment map
  fake reflection, creating with 347-354

F

fbxLoader
  used, for visualizing motions captured models 310-313
FBX model 308
Field of View (FOV) 59
file formats, Three.js
  3DM 241
  3DS 242
  3MF 241
  AMF 241
  COLLAborative Design Activity (COLLADA) 241
  Draco 241
  GCode 241
  glTF 241
  Industry Foundation Classes (IFC) 241
  JSON 241
  KMZ 241
  LDraw 241
  LWO 242
  MTL 242
  NRRD 242
  OBJ 242
  Packed Raw WebGL Model (PRWM) 242
  PCD 242
  Polygon File Format (PLY) 242
  Protein Data Bank (PDB) 242
  STereoLithography (STL) 242
  SVG 242
  TILT 242
  Virtual Reality Modeling Language (VRML) 243
  Visualization Toolkit (VTK) 243
  VOX 243
  XYZ 243
FilmPass
  creating 375
  parameters 375
Firefox plugin
  download link 508
FirstPersonControls 283, 290-292
fixed joint 428
  two objects, connecting with 428-431
FlyControls 283, 289, 290
Fork Colored Flickering Stars
  reference link 138
frames per second (FPS) 24
friction
  working with 421-425

G

GCode 241
  URL 241
geometries
  combining, with meshes 45
  creating, by extruding 2D shape 187
  creating, through Constructive Solid Geometry (CSG) 240
  loading, from external resources 240
  merging 238-240
  properties and functions 45-52
geometries, for debugging
  THREE.EdgesGeometry 199, 200
  THREE.WireFrameGeometry 200, 201
gizmos 284
GLSL 134
glTF 240, 241
  reference link 241
gltfLoader
  using 308-310
glTF model 308
  loading 252-254

H

HDR images
  loading, as texture 324
heads-up display (HUD) 225
helper functions 30, 31
helper objects 30, 31
HTML5 Canvas 3
HTML5 tags 3
HTML structure
  exploring, for Three.js application 14, 15
HTML webpack plugin
  reference link 15

I

Industry Foundation Classes (IFC) 241

J

JavaScript 3
joints 428
  fixed joint 428
  prismatic joint 428
  revolute joint 428
  spherical joint 428
  using, to limit object movement 428
JSON 241

K

kaleidoscope effect 395
KMZ 241, 315
  reference link 241

L

LDraw 241
  URL 256
LEGO models
  displaying 254-256
lighting
  adding, to scene 466-469
lightmap
  baking, in Blender 464
  fake lighting, creating with 334-336
lights 19, 33
  THREE.AmbientLight 19
  THREE.DirectionalLight 19
light textures
  baking 469-475
lil-gui library
  reference link 28
line geometry
  materials, using for 140
loaders
  examples 264
LuminosityHighPassShader effect 397
LWO 242
  URL 242

M

MagicaVoxel 256
  reference link 256
material properties 104
  advanced properties 104, 106, 107
  basic properties 104-106
  blending properties 104, 106
materials
  texture, using 319
  using, for line geometry 140

materials, for line geometry
  THREE.LineBasicMaterial 141, 142
  THREE.LineDashedMaterial 144, 145
MD2 model 308
meshes 16, 34
  animating 24, 25
  combining, with geometries 45
  functions 52-54
  location, setting with position property 54
  position, modifying with translate property 56, 57
  properties 52-54
  rotation, defining with rotation property 54-56
metalness texture
  applying, to model 336, 337
MIP map 322
MirrorShader effect 396
Mixamo
  URL 308
mixer and morph targets
  animating with 297-300
morph targets 294
  animating with 296
MTL 242, 248-252
multiple materials
  using, for single mesh 119-123

N

Node.js
  reference link 11
normal map
  detailed bumps and wrinkles, achieving with 326-329
Notepad++ 7
  reference link 7
NRRD 242
  reference link 242

O

OBJ 242, 248-252
objects
  dragging 273-277
  grouping 233-238
  selecting 271-273
OpenGL Shading Language (GLSL) 401
  reference link 401
OpenGL Wiki
  reference link 106
orbit controls
  enabling 26, 27
OrbitControls 19, 283, 292, 294
orthographic camera
  properties 61-63
  versus perspective camera 57, 59

P

Packed Raw WebGL Model (PRWM) 242
  reference link 242
parallax barrier 57
particles
  image, drawing on canvas 215-219
  styling, with textures 215, 219-225
PCD
  reference link 242
Peano-Gosper Curve
  reference link 142
perspective camera
  properties 59, 60
  versus orthographic camera 57-59
PLY model
  point cloud, loading from 262-264

PointerLockControls 283
points 212-214
  configuring 214
Polygon File Format (PLY) 242
polyhedron 171
positional audio 439
position property
  used, for defining rotation of mesh 54-56
  used, for modifying position of mesh 56, 57
  used, for setting location of mesh 54
postprocessing passes, Three.js 373, 374
  AdaptiveToneMappingPass 372
  additional passes 381, 382
  bloom effect, adding to scene with THREE.BloomPass 377, 378
  BloomPass 372
  BokehPass 372
  ClearPass 372
  CubeTexturePass 372
  DotScreenPass 372
  FilmPass 372
  GlitchPass 372
  HalfTonePass 372
  LUTPass 372
  MaskPass 372
  multiple renderers output, displaying on same screen 380
  OutlinePass 372
  RenderPass 372
  SAOPass 373
  SavePass 373
  scene, outputting as set of dots 378, 379
  ShaderPass 373
  SMAAPass 373
  SSAARenderPass 373
  SSAOPass 373
  SSRPass 373
  TAARenderPass 373
  TexturePass 373
  TV-like effect, creating with THREE.FilmPass 375-377
  UnrealBloomPass 373
prismatic joint 428
  movement, limiting to single axis 437, 438
Protein Data Bank (PDB) 242
  proteins, displaying from 258-262
  URL 258

Q

Quake model
  animation, loading from 313

R

Rapier
  documentation link 412
  dominos, simulating 415-421
  reference link 412
  rigid body types 414
  supported shapes 425-427
  Three.js scene, creating with 410, 411
raycaster 272
React.js 14
renderer 34
render loop
  updating 370
repeat wrapping 355-357
requestAnimationFrame 23, 24
restitution
  working with 421-425
revolute joint 428
  rotation, limiting with 433-436
RGBShiftShader effect 396
roughness texture
  applying, to model 338, 339

S

scene
  background, modifying 41-43
  creating 33, 34
  creating, by using different cameras 57
  exporting, from Blender 475-478
  exporting, from Three.js 444-450
  fog, adding 39, 40
  functionality 34-36
  importing, into Blender 444-478
  lighting, adding 466-469
  materials, updating 43-45
  objects, adding and removing 37-39
  setting up, in Blender 464-466
shaderpass-blurs.html example 393
shaderpass-simple.html example 392, 393
shaders
  reference link 369
Shadertoy
  URL 399
simple animations 268-270
simple integrated server
  experimenting 11-13
  testing 11-13
simple materials
  combining 114, 115
  multiple materials, using for single mesh 119-123
  THREE.MeshBasicMaterial 108-112
  THREE.MeshDepthMaterial 112, 113
  THREE.MeshNormalMaterial 116, 117
  working with 107, 108
simple shaders
  working with 394
simulating lighting, WebGL
  reference link 67
skeleton
  visualizing, with BVHLoader 316, 317
skeleton animation 295
SobelOperatorShader effect 395
special lights
  THREE.HemisphereLight 90, 91
  THREE.LensFlare 96-99
  THREE.LightProbe 93-96
  THREE.RectAreaLight 91-93
  working with 89
specular map
  using, to determine shininess 344-347
spherical joint 428
  objects, connecting with 431-433
sprite.html 210
sprite maps
  working with 226-228
sprites 210, 211
  configuring 212
static scene
  exporting, from Blender 450-454
  importing, into Three.js 450-454
stats.js library 23
stereo effect 57
STereoLithography (STL) 242
Sublime Text Editor 7
  reference link 7
SVG 192, 242
SVG element
  3D shapes, extruding from 192-195

T

TeapotGeometry 186, 187
  properties 187
texels 321

texture
  applying, to mesh 320-323
  HDR images, loading as 324
  loading 320-323
  output, using from video as 362, 363
  used, for styling particles 215-225
  using, in materials 319
THREE.AmbientLight 68-72
THREE.AnimationAction 297
THREE.AnimationClip 297, 300-303
THREE.AnimationMixer 297, 301, 302
THREE.BloomPass
  used, for adding bloom effect to scene 377, 378
THREE.BoxGeometry 160, 161
THREE.CircleGeometry 151-153
  properties 152
THREE.Color object 87-89
THREE.ConeGeometry 166
  properties 166, 167
THREE.ConvexGeometry 180-182
THREE.CylinderGeometry 164, 165
THREE.DataTexture
  using 362
THREE.DirectionalLight 68, 84-87
THREE.DodecahedronGeometry 176, 177
THREE.EdgesGeometry 199, 200
THREE.EffectComposer
  configuring, for postprocessing 370
  creating 369
THREE.ExtrudeGeometry
  properties 189
  using 187-190
THREE.FilmPass
  used, for creating TV-like effect 375-377
THREE.Group
  using 236
THREE.HemisphereLight 68, 90, 91
THREE.IcosahedronGeometry 173
Three.js 3
  animation, importing into 454-463
  AR, using 514-516
  postprocessing passes 372-375
  render loop, updating 370
  scene, exporting from 444-450
  setting up, for postprocessing 367, 368
  static scene, importing into 450-454
  using, with TypeScript 488-493
  VR, using 508-514
Three.js and React
  using, with React Three Fiber 499-507
  using, with TypeScript 493-499
Three.js animation
  reference link 3
Three.js application
  HTML structure, exploring for 14, 15
Three.js editor
  reference link 7
Three.js JSON format
  loading 243-247
  saving 243-247
Three.js object
  rendering 414, 415
Three.js scene
  creating, with Rapier 410-412
  sound sources, adding to 438-440
Three.js scene, creating with Rapier 410-412
  world, creating 412-414
Three.js source code
  archive, downloading 10
  archive, extracting 10
  Git repository clone, using 9
  obtaining 9
THREE.LatheGeometry 182, 183
  properties 183
THREE.LensFlare 68, 96-99

THREE.LightProbe 68, 93-96
THREE.LineBasicMaterial 141, 142
  properties 141
THREE.LineDashedMaterial 144, 145
THREE.MeshBasicMaterial 108-112
  properties 108
THREE.MeshDepthMaterial 112, 113
THREE.MeshLambertMaterial 124-126
  properties 124
THREE.MeshNormalMaterial 116, 117
THREE.MeshPhongMaterial 127
  properties 127
THREE.MeshPhysicalMaterial 131
  properties 131
THREE.MeshStandardMaterial 129, 130
  properties 129
THREE.MeshToonMaterial 128, 129
THREE.OctahedronGeometry 175, 176
THREE.ParametricGeometry
  arguments 197
  using 195-199
THREE.PlaneGeometry 149, 150
  properties 150
THREE.PointLight 68, 80, 81
  properties 82-84
THREE.Points object
  creating, from existing geometry 228-230
THREE.PolyhedronGeometry 171-173
THREE.RectAreaLight 68, 91-93
THREE.RingGeometry 153
  properties 154
THREE.RoundedBoxGeometry 185
  properties 185
THREE.ShaderMaterial
  properties 133
  shaders, using with 133-139
THREE.ShaderPass
  blurring shaders 397-399
  shaderpass-blurs.html example 393
  shaderpass-simple.html example 392
  simple shaders 394
  used, for custom effects 392, 393
THREE.ShadowMaterial 132
THREE.ShapeGeometry 155-159
THREE.SphereGeometry 161-163
THREE.SpotLight 68, 72, 77-80
  properties 73-76
THREE.TetrahedronGeometry 174
THREE.TorusGeometry 167
  arguments 168
THREE.TorusKnotGeometry 169, 170
THREE.TubeGeometry
  arguments 191
  using 190, 191
THREE.WireFrameGeometry 200, 201
TILT 242
  URL 242
torus 167
TrackballControls 283, 286-288
transparent models
  creating, with alpha map 339-341
Troika library
  used, for creating text 205-207
tweening 277
Tween.js 14, 277
  animating with 277-282
TypeScript 3, 488
  Three.js and React, using with 493-499
  Three.js, using with 488-493

U

useCallback 497
useRef 497
UV mapping 323

UV modeling
  in Blender 480-486

V

vertex and fragment shaders
  creating 399
Virtual Reality Modeling Language (VRML) 243
virtual reality (VR) 3
Visualization Toolkit (VTK) 243
Visual Studio Code 7
  reference link 7
Vite
  reference link 488
VOX 243
  URL 243
voxel-based models
  loading 256-258
VR headset
  simulating 509

W

WebGL 3
  URL 399
webpack guide
  reference link 15
WebStorm 7
  reference link 7
WebXR Device API
  reference link 514

X

X3D models
  URL 243
XYZ
  reference link 243

Hi!

I am Jos Dirksen, author of Learn Three.js Fourth Edition. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in Three.js.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on Learn Three.js Fourth Edition.

Go to the link below or scan the QR code to leave your review:

`https://packt.link/r/1803233877`

Your review will help me to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,

Jos Dirksen

`Packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

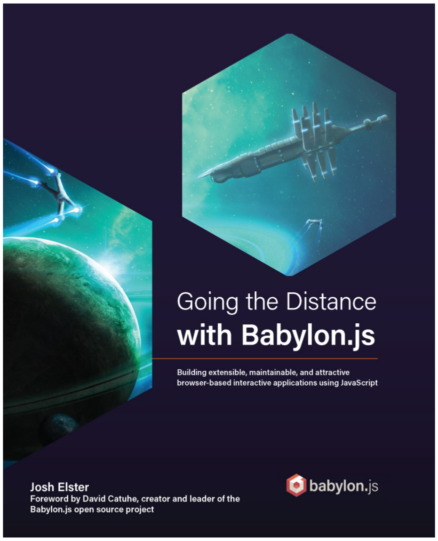
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `packt.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

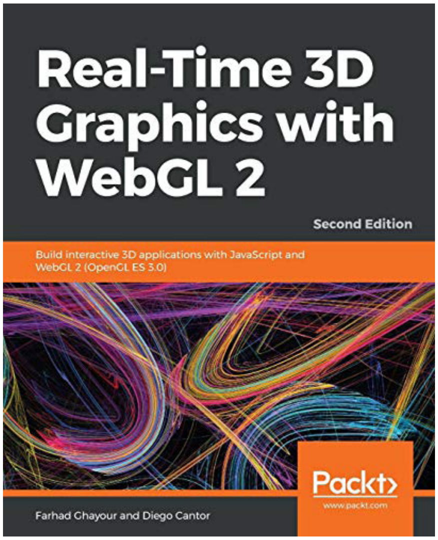

Going the Distance
with Babylon.js
Building extensible, maintainable, and attractive browser-based interactive applications using JavaScript
Josh Elster
Foreword by David Catuhe, creator and leader of the Babylon.js open source project
babylon.js

Going the Distance with Babylon.js

Josh Elster

ISBN: 9781801076586

- Use Babylon.js v5.0 to build an extensible open-source 3D game accessible with a web browser
- Design and integrate compelling and performant 3D interactive scenes with a web-based application
- Write WebGL/WebGPU shader code using the Node Material Editor
- Separate code concerns to make the best use of the available resources


Real-Time 3D Graphics with WebGL 2
Second Edition
Build interactive 3D applications with JavaScript and WebGL 2 (OpenGL ES 3.0)
Packt>
www.packt.com
Farhad Ghayour and Diego Cantor

Real-Time 3D Graphics with WebGL 2

Farhad Ghayour | Diego Cantor

ISBN: 9781788629690

- Understand the rendering pipeline provided in WebGL
- Build and render 3D objects with WebGL
- Develop lights using shaders, 3D math, and the physics of light reflection
- Create a camera and use it to navigate a 3D scene
- Use texturing, lighting, and shading techniques to render realistic 3D scenes
- Implement object selection and interaction in a 3D scene

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803233871`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly